COP 4610

Operating System Principles

Lecture 4 – Processes

1

Recap – Last Lecture

- Operating System Services
- System Calls and Interrupts
- System Programs
- Operating System Design and Implementation
- System Layering
- Virtual Machines

2

Overview

- Process Concepts
- Process Scheduling
- Operations on Processes
- Inter-process Communication

3

3

Process Concept

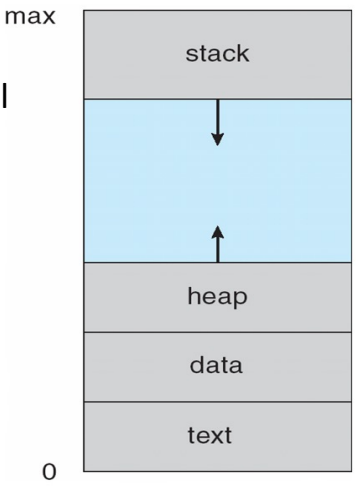
- Job/**Process**/Task used interchangeably
- A process is an instance of a program ("program in execution")
- Program: "piece of code"
- Process: code + data + more data + control structures + …

4

4

Process Memory Layout

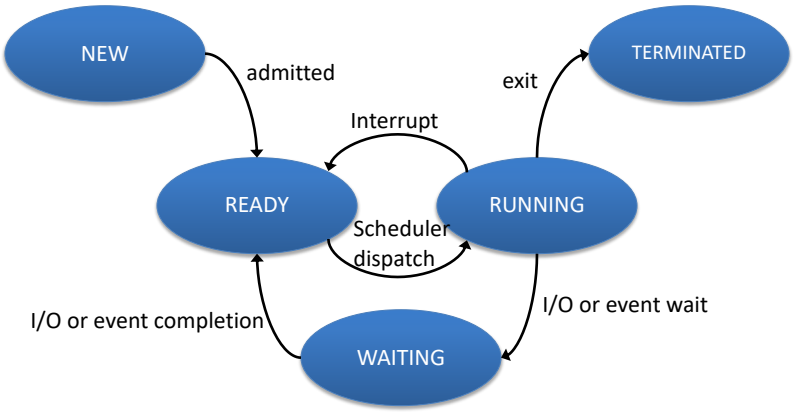
- **Stack**: Temporary Data
  - Function parameters, local variables, function call frames.
- **Heap**: Dynamically allocated memory.
- **Data**: Global variables & constant strings.
- **Text**: Program code.



5

5

Process "Life Cycle"



6

6

Process State

- A process changes **state** constantly:
  - **New**: being created.
  - **Running**: running on a processor core.
  - **Waiting**: waiting for an *event*.
  - **Ready**: (or **runnable**) waiting for a core.
  - **Terminated**: finished/dead.

7

7

OS Information about Processes

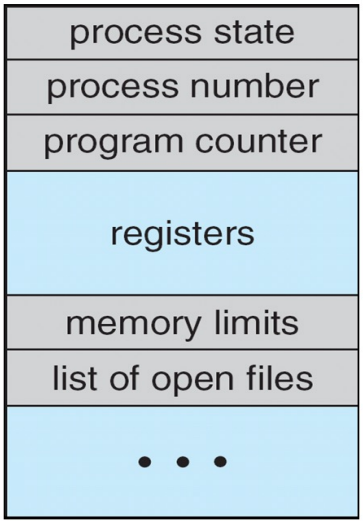
- Memory (Stack, Heap, Code, Static Data)
- Current State (e.g., program counter)
- Process ID
- Saved Registers
- Open Files
- Other bookkeeping data

This is called the **process control block (PCB)** or **task control block (TCB)**.

8

8

Process Control Block



9

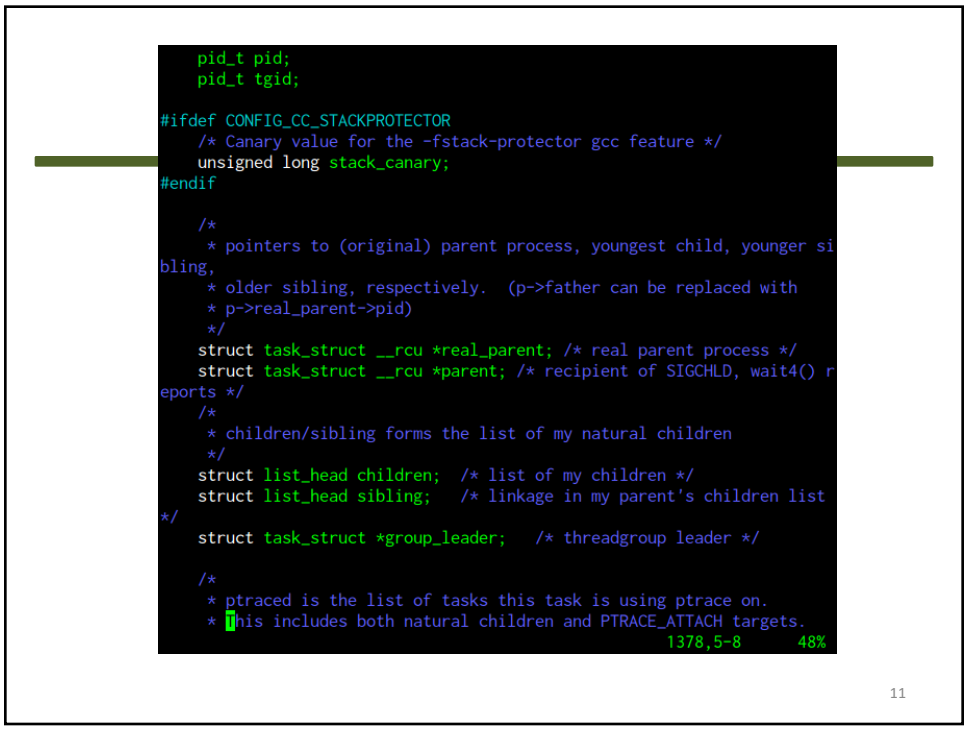
9

PCB on Linux

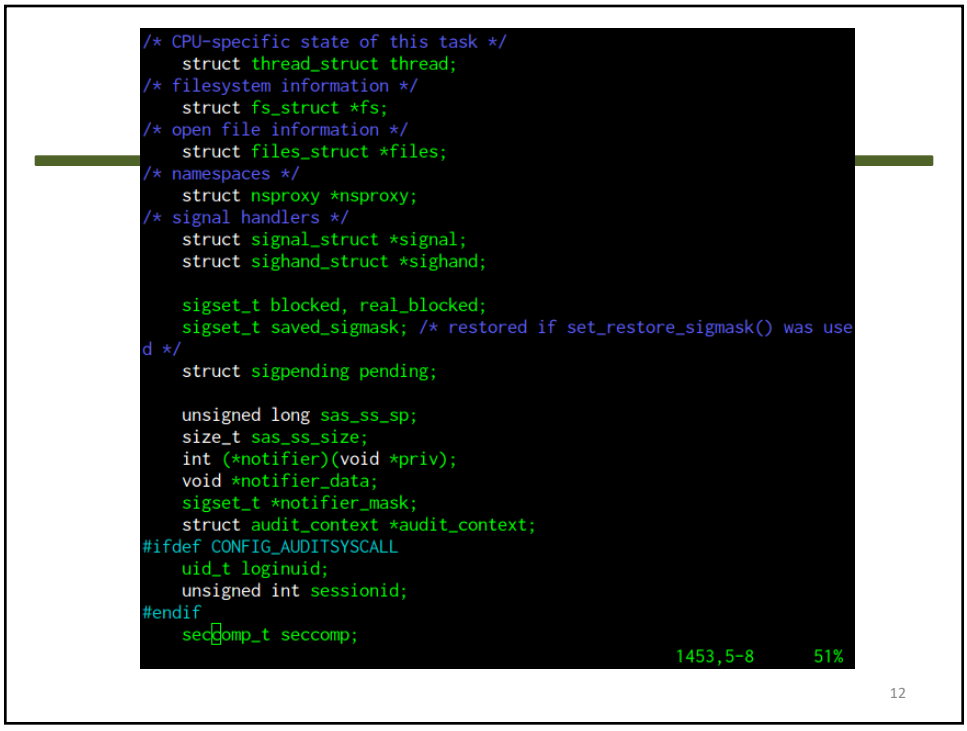
```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
#endif
    int on_rq;

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
#endif

    /*
     * fpu_counter contains the number of consecutive context switches
     * that the FPU is used. If this is over a threshold, the lazy fpu
     * saving becomes unlazy to save the trap. This is an unsigned char
                                                    1287,0-1      45%
```

10

10

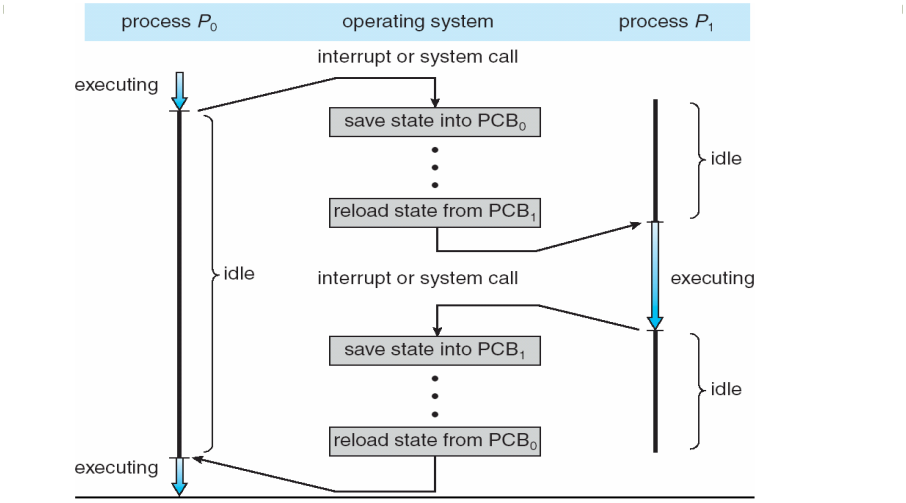

pid_t pid;
pid_t tgid;
#ifdef CONFIG_CC_STACKPROTECTOR
/* Canary value for the -fstack-protector gcc feature */
unsigned long stack_canary;
#endif
/*
* pointers to (original) parent process, youngest child, younger si
bling,
* older sibling, respectively. (p->father can be replaced with
* p->real_parent->pid)
*/
struct task_struct __rcu *real_parent; /* real parent process */
struct task_struct __rcu *parent; /* recipient of SIGCHLD, wait4() r
eports */
/*
* children/sibling forms the list of my natural children
*/
struct list_head children; /* list of my children */
struct list_head sibling; /* linkage in my parent's children list
*/
struct task_struct *group_leader; /* threadgroup leader */
/*
* ptraced is the list of tasks this task is using ptrace on.
* This includes both natural children and PTRACE_ATTACH targets.
1378,5-8 48%
11

11


/* CPU-specific state of this task */
struct thread_struct thread;
/* filesystem information */
struct fs_struct *fs;
/* open file information */
struct files_struct *files;
/* namespaces */
struct nsproxy *nsproxy;
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked;
sigset_t saved_sigmask; /* restored if set_restore_sigmask() was use
d */
struct sigpending pending;
unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
struct audit_context *audit_context;
#ifdef CONFIG_AUDITSYSCALL
uid_t loginuid;
unsigned int sessionid;
#endif
seccomp_t seccomp;
1453,5-8 51%
12

12

Process Switching ("Context Switch")



13

13

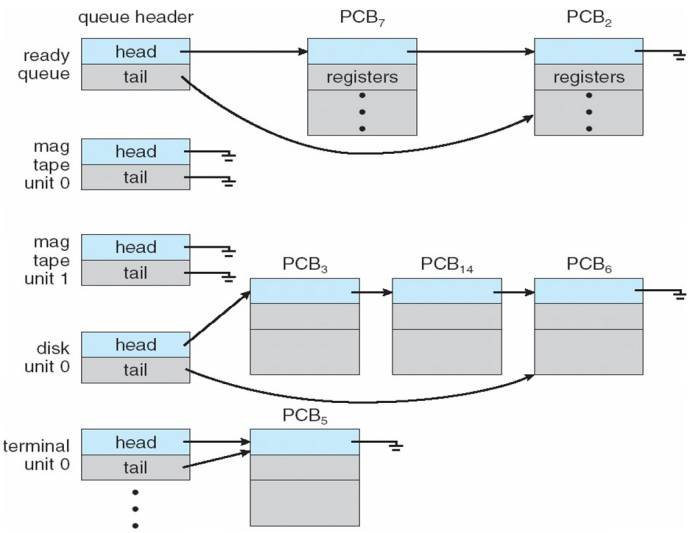
Process "Wait" Queues

- **Job Queue** is all processes on the system.
- **Ready Queue** (**Run Queue**) contains processes waiting to run, or **waiting for the CPU**!
- **Device Queue** processes waiting for a device event ("blocked" devices).
- **"Other" Queues** contain processes waiting for other processes to finish, sleeping for time, etc.

14

14

Process "Wait" Queues



15

Schedulers

- **Long-term scheduler** which processes should be run in the future?
  - Degree of multiprogramming!
- **Short-term scheduler** which process should be run next?
  - Manages queues and quickly decides next process to run.

16

16

Scheduling Concerns

- Is enough RAM available to satisfy running processes?
- Is device throughput able to support more IO-bound processes?
- Is there enough CPU time available to satisfy all processes? (**long**) How do I schedule fairly? (**short**)
- **Are there benefits for sleeping (swapping) a process for an extended time?** (**long**)

17

17

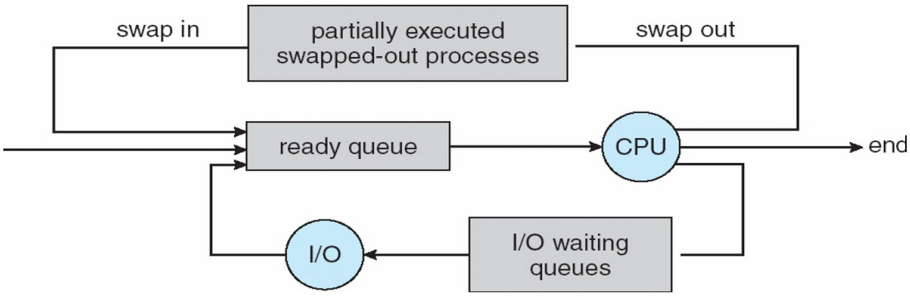
Schedulers

- Short-term: invoked frequently (milliseconds); must be fast
- Long-term: infrequently (seconds)
- **I/O-bound process:** spends more time doing I/O than processing (CPU bursts can be frequent, but are short)
- **CPU-bound:** spends more time doing computations (very long CPU bursts)

18

18

"Medium" Scheduling



19

19

Process Creation

- A process is always created via a parent, except for process **1**, **/sbin/init**.
- A parent can have multiple children. Entire structure is a **tree**.
- Each process has a unique identifier, the **process identifier**, or **pid** (get the pid of a process using the getpid() system call).

20

20

```
PSTREE(1)                  User Commands                  PSTREE(1)

NAME
       pstree - display a tree of processes

SYNOPSIS
       pstree [-a|--arguments] [-c|--compact] [-h|--high
       light-all|-Hpid|--highlight-pid pid] [-g|--show-pgids]
       [-l|--long] [-n|--numeric-sort] [-p|--show-pids]
       [-s|--show-parents] [-u|--uid-changes] [-Z|--security-context]
       [-A|--ascii|-G|--vt100|-U|--unicode] [pid|user]
       pstree -V|--version

DESCRIPTION
       pstree shows running processes as a tree. The tree is rooted at
       either pid or init if pid is omitted. If a user name is  speci
       fied,  all process trees rooted at processes owned by that user
       are shown.

       pstree visually merges identical branches by  putting  them  in
       square  brackets  and prefixing them with the repetition count,
       e.g.

           init-+-getty
                |-getty
                |-getty
                `-getty

       becomes

           init---4*[getty]

       Child threads of a process are found under the  parent  process
 Manual page pstree(1) line 1 (press h for help or q to quit)
```

21

21

```
[pdonnel3@ccl12 ~]$ pstree | head -n 20
init-+-NetworkManager-+-dhclient
     |                `~{NetworkManager}
     |-abrt-dump-oops
     |-abrtd
     |-acpid
     |-atd
     |-auditd-+-audispd-+-sedispatch
     |        |         `-{audispd}
     |        `~{auditd}
     |-automount---4*[{automount}]
     |-avahi-daemon---avahi-daemon
     |-cachefilesd
     |-cimserver---cimservermain---4*[{cimservermain}]
     |-condor_master-+-chirp_server
     |               |-condor_mouse_wa
     |               |-condor_schedd
     |               `-condor_startd---3*[condor_starter---condor_exec.exe---work_queue_work]
     |-crond
     |-cupsd
     |-dbus-daemon
[pdonnel3@ccl12 ~]$
```

22

22

23

Process Creation

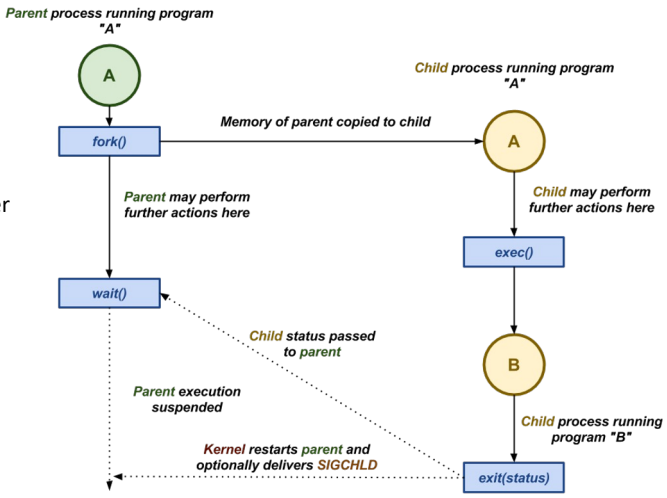
- **Resource sharing**
  1. Parent and children share all resources
  2. Children share subset of parent's resources
  3. Parent and child share no resources
- **Execution**
  1. Parent and children execute concurrently
  2. Parent waits until children terminate
- **Address space**
  1. Child is duplicate of parent
  2. Child has a program loaded into it

24

24

Process: Life Cycle

1. **Parent forks** to create a new process
2. **Child** performs actions, possibly **exec** to run another program
3. **Parent waits** for **child** process
4. **Child exits**
5. **Parent** receives **child**'s exit status



25

Example Code

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
   pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
   }
   else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
   }
   return 0;
}
```

26

26

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call
  - Returns status data from child to parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Terminate with **abort()**: **SIGABRT** signal, can be caught; core dump
- Terminate a process from another process: **kill()**: **SIGKILL** (also "kill -9 pid" from terminal)
- Parent may terminate the execution of children for various reasons:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

27

27

Signals

Signals are a means of asynchronously notifying a **process** of an **event**.

- Each **signal** delivers a small integer that represents a particular **event**
- To deliver the **event**, the **kernel** will interrupt the normal execution of the **process**
- **Processes** register **handlers** to catch certain **events**
- After the **handlers** are executed, the **process** will continue executing where it was interrupted

28

28

Signal: "Kill"

Kill

Send a **signal** to a **process**

```
kill(pid, SIGTERM);
```

Signal

Register a callback function for particular **event**

```
void handler(int signum) {
        puts("Handler");
}

signal(SIGTERM, handler);
```

29

29

SIGCHLD / SIGALRM

SIGCHLD

When a **child** process exits, the **parent** is notified with the **SIGCHLD** event

```
signal(SIGCHLD, handler);
```

SIGALRM

An alarm or timer can be set by first using **alarm**, and then handling the **SIGALRM** event when it is triggered

```
signal(SIGALRM, handler);
alarm(5);
```

30

30

Process Termination

- What happens if a process "dies"?
  - The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process: `pid = wait(&status);`
  - Parent "collects" child's resources
  - If no parent waiting (did not invoke `wait()`) process is a **zombie**
- What happens if parent "dies"?
  - If parent terminated without invoking `wait`, process is an **orphan**
  - May receive "new parent" (grandparent, init process, etc.)
  - **Cascading termination**: no orphans allowed; if a process terminates, all its children must also be terminated

31

31

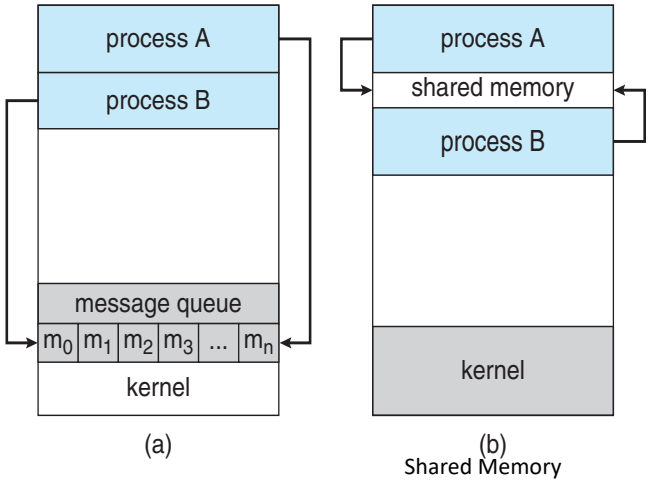
Interprocess Communication

- Processes **communicate** by sharing data.
- Why do processes communicate?
  - Computation speedup
  - Modularity
  - Information sharing
- Mechanism: **interprocess communication** (**IPC**)
- Two standard models: **Shared Memory** and **Message Passing**

32

32

Communication Models



33

33

Producer-Consumer Problem

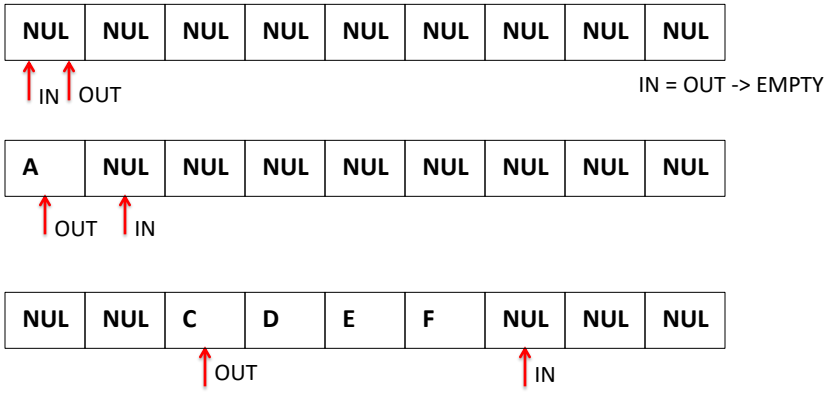
- One process produces data. The second process consumes the data.
- Data stored in a **buffer**:
  - **Unbounded-Buffer** has no limit on size. Grows to size of memory.
  - **Bounded-Buffer** has fixed size. Creates a new problem:
    - **How do we handle the producer creating data too fast?**

34

34

Shared Memory Solution

Circular buffer



35

35

Shared Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

36

36

Bounded Buffer - Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

37

37

Bounded Buffer - Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */

        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

38

38

Message Passing

- Two primitives:
  - **send** (C, message) – messages have maximum size
  - **receive** (P, message)
- Think **mailboxes.**
- **Kernel** usually manages the message passing and “mailboxes”.

39

39

Message Passing Considerations

- How is the link established?
  - Automatically on send?
- Can the link be asymmetric?
  - Receiving a message: who is the sender?
- Is there a limit to the capacity of the link?
- Is the message size fixed or variable?
- Is a link unidirectional or bidirectional?
- Can there be multiple links between a pair of communication processes?

40

40

Message Passing

- Direct Communication
  - send (P, message) -> receiver process P
  - receive (Q, message) -> sender process Q
- Indirect Communication ("mailboxes")
  - send (M1, message) -> put in mailbox M1
  - receive (M1, message) -> take from mailbox M1

41

41

IPC Synchronization

- **Blocking**?
  - **Consumer** is put in a waiting scheduler queue if "mailbox" is empty.
  - **Producer** is put in a waiting scheduler queue if "mailbox" is full.
- **Non-blocking?**
  - Neither **Producer** nor **Consumer** blocks; failure is returned from message passing primitive instead.

42

42

Buffering

- Queue of messages attached to the link; implemented in one of three ways:
  - **Zero capacity** – no messages are queued on a link. Sender must wait for receiver (rendezvous).
  - **Bounded capacity** – finite length of *n* messages. Sender must wait if link full.
  - **Unbounded capacity** – infinite length. Sender never waits.

43

IPC - POSIX

- POSIX Shared Memory

```
shm_id = shm_open(name, O CREAT | O RDWR, 0666);

ftruncate(shm_id, 4096);

shared_memory = (char *) shmat(shm_id, NULL, 0);

sprintf(shared_memory, "Writing to shared
memory");
```

Also: shmdt (remove), **shmctl** (destroy)

44

IPC - Mach

- Mach communication is message based
  - Even system calls are messages
  - Each task gets two mailboxes at creation- Kernel and Notify
  - Only three system calls needed for message transfer

    `msg_send(), msg_receive(), msg_rpc()`
  - Mailboxes needed for communication, created via

    `port_allocate()`

45

45

Recap

- Key Points:
  - System Layering
  - Concept of a Process
  - Scheduling
  - Process Creation and Termination
  - Interprocess Communication

46

46