Chapter 3: Processes Concept

Prof. Li-Pin Chang
CS@NYCU

Chapter 3: Processes-Concept

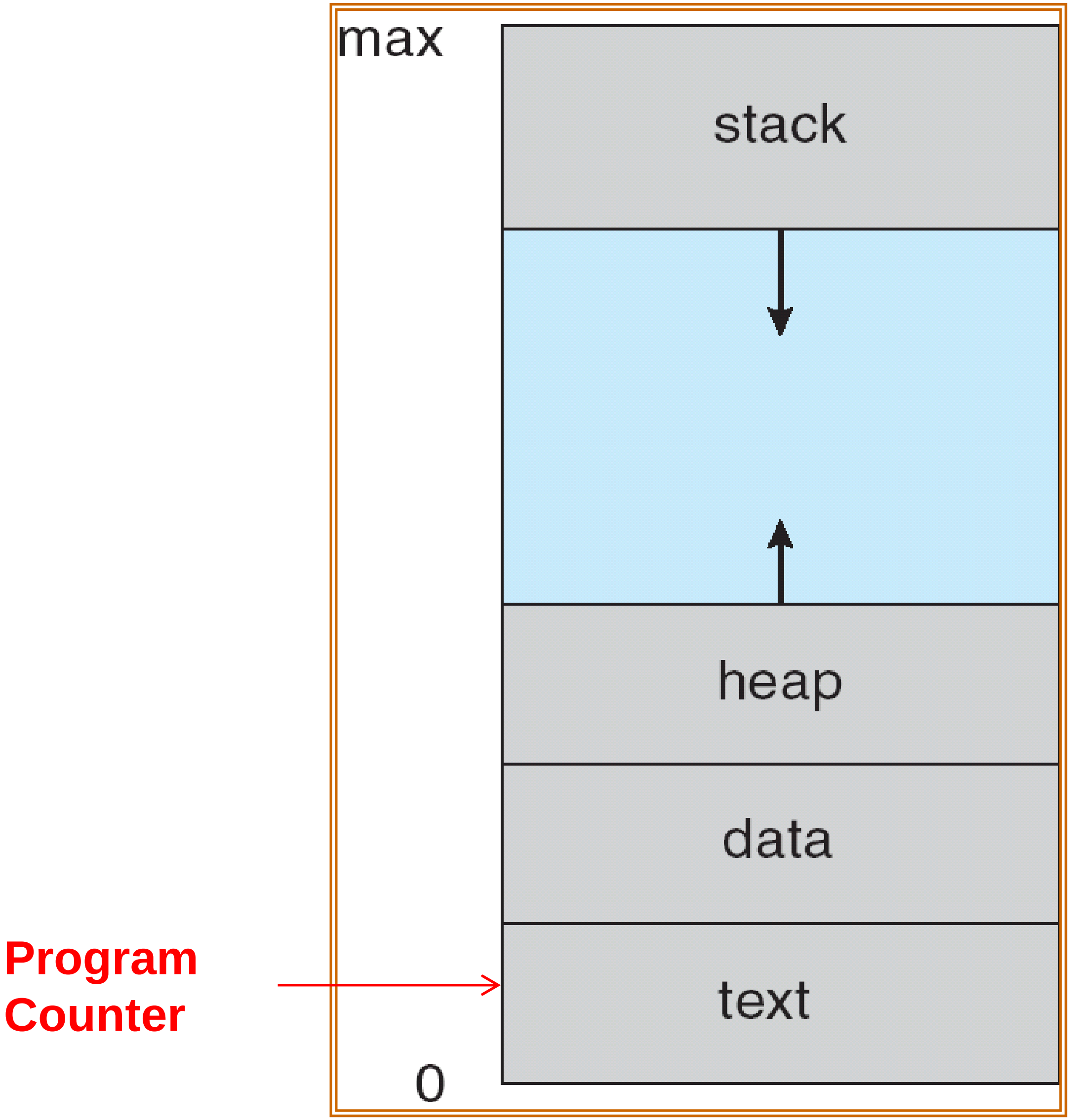
- Process Concepts
- Types of process Schedulers
- Operations on Processes
- Inter-process Communication
- Examples of IPC

PROCESS CONCEPTS

Process Concept

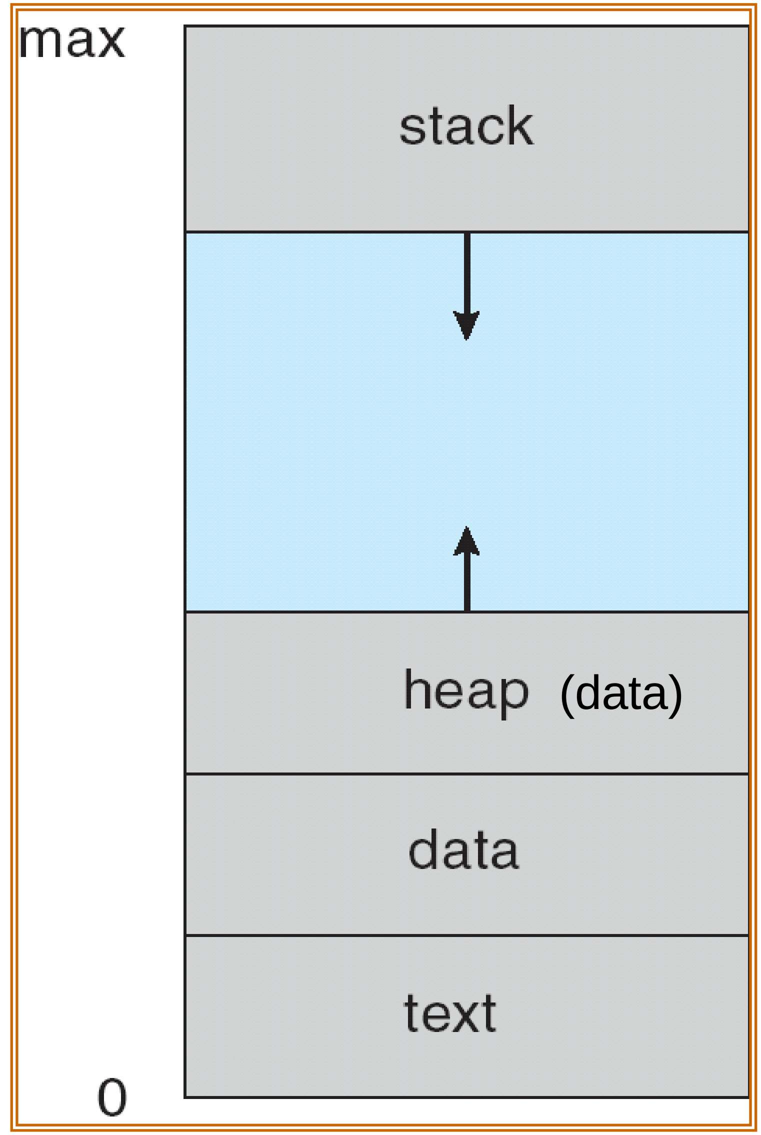
- An operating system executes a variety of programs
  - We use the terms job, task, and process interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
  - Process: active,   program: passive
- A process uses the following context
  - Text section: executable binaries
  - Stack section: function args + local vars
  - Data section: global vars (w/o init values → BSS) + heap
  - Program counter and other CPU registers

Process in Memory (virtual addr space)



Where the variables below are allocated from?

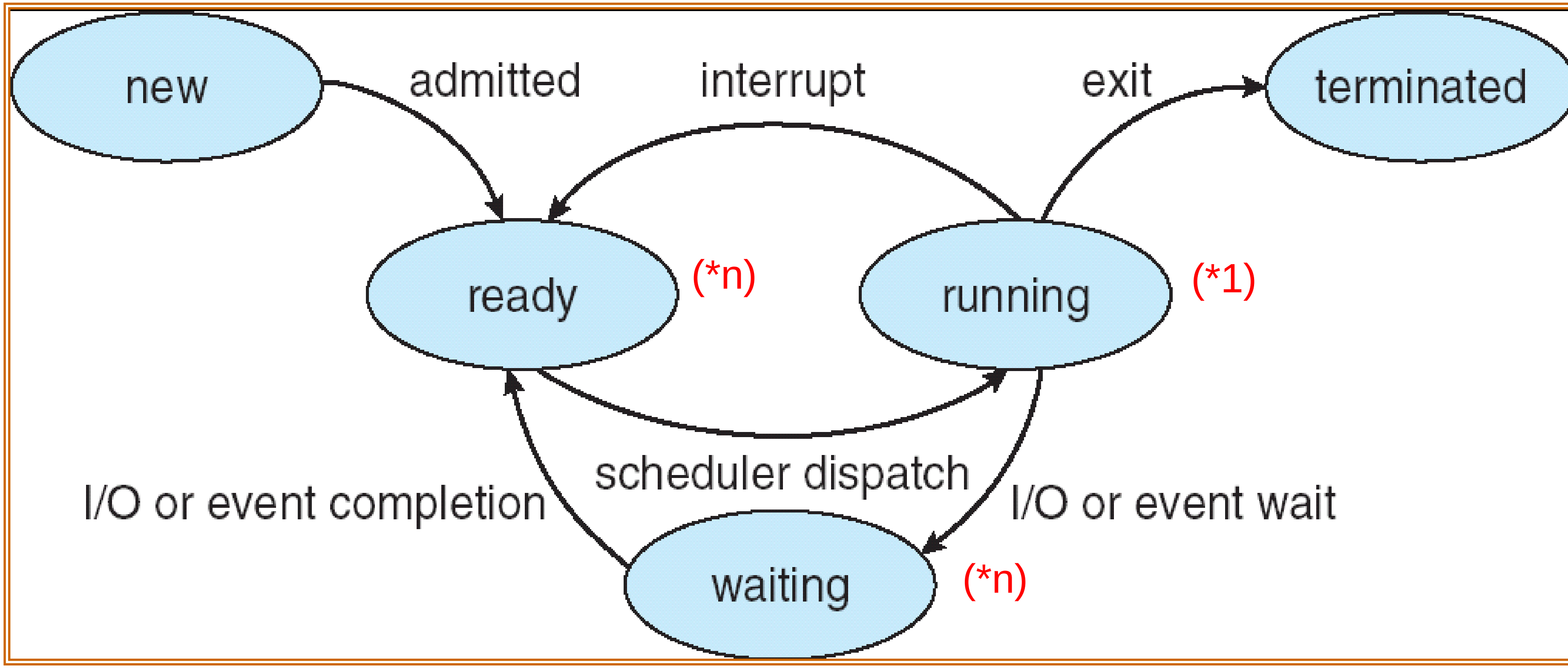
```
int i,j=2;

int foo(int x)
{
	int *y;
	static char c='x';

	i=0;
	y=(int *)malloc(100);
}
```



Process State

- As a process executes, it changes state
  - new: The process is being created
  - running: Instructions are being executed
  - ready: The process is waiting to be assigned to a processor
  - waiting: The process is waiting for some event to occur
  - terminated: The process has finished execution

Diagram of Process State



Running → waiting or ready

A running process voluntarily leaves the running state

- Running → waiting: the running process requests and waits on a system service that can not be immediately fulfilled
  - Involveing a trap (initiating a synchronous I/O)

A running process involuntarily leaves the running state

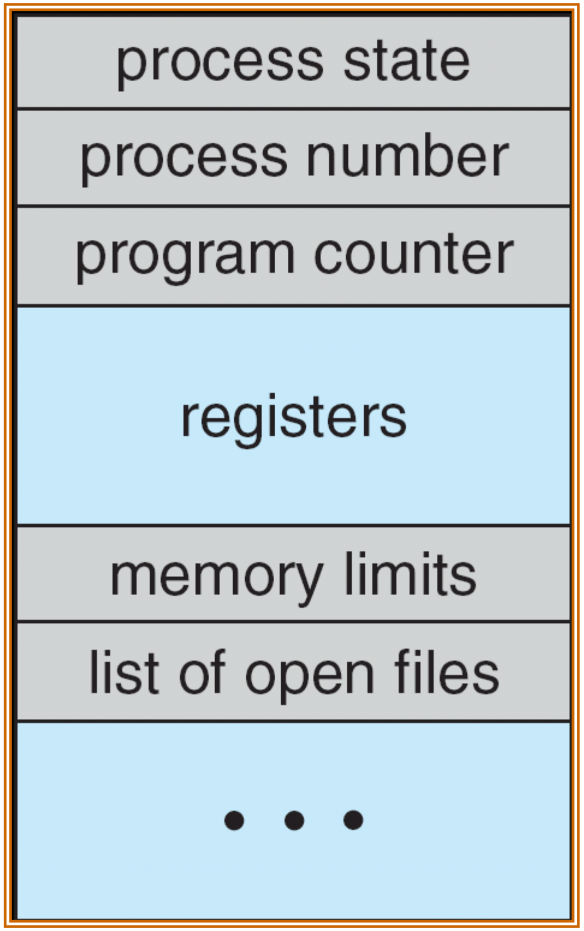
- Running → ready, case 1: the running process runs out its time quantum under time sharing
  - Triggered by a timer interrupt
- Running → ready, case 2: IO interrupts make a high-priority process ready and the running process is preempted by the high-priority one
  - Triggered by an I/O interrupt

- Which one(s) of the following transitions can be triggered by a hardware interrupt?
    1. Running → ready
    2. Running → waiting
    3. Waiting → ready

- What is the process state transition of starting an synchronous I/O and resuming execution after it?

Process Control Block (PCB)

- Information associated with each process
  - Process state
  - Saved CPU registers values
  - CPU scheduling info (e.g., priority)
  - Memory-management information (e.g., segment table and page-table base register)
  - I/O status info (e.g., opened files)
  - Etc

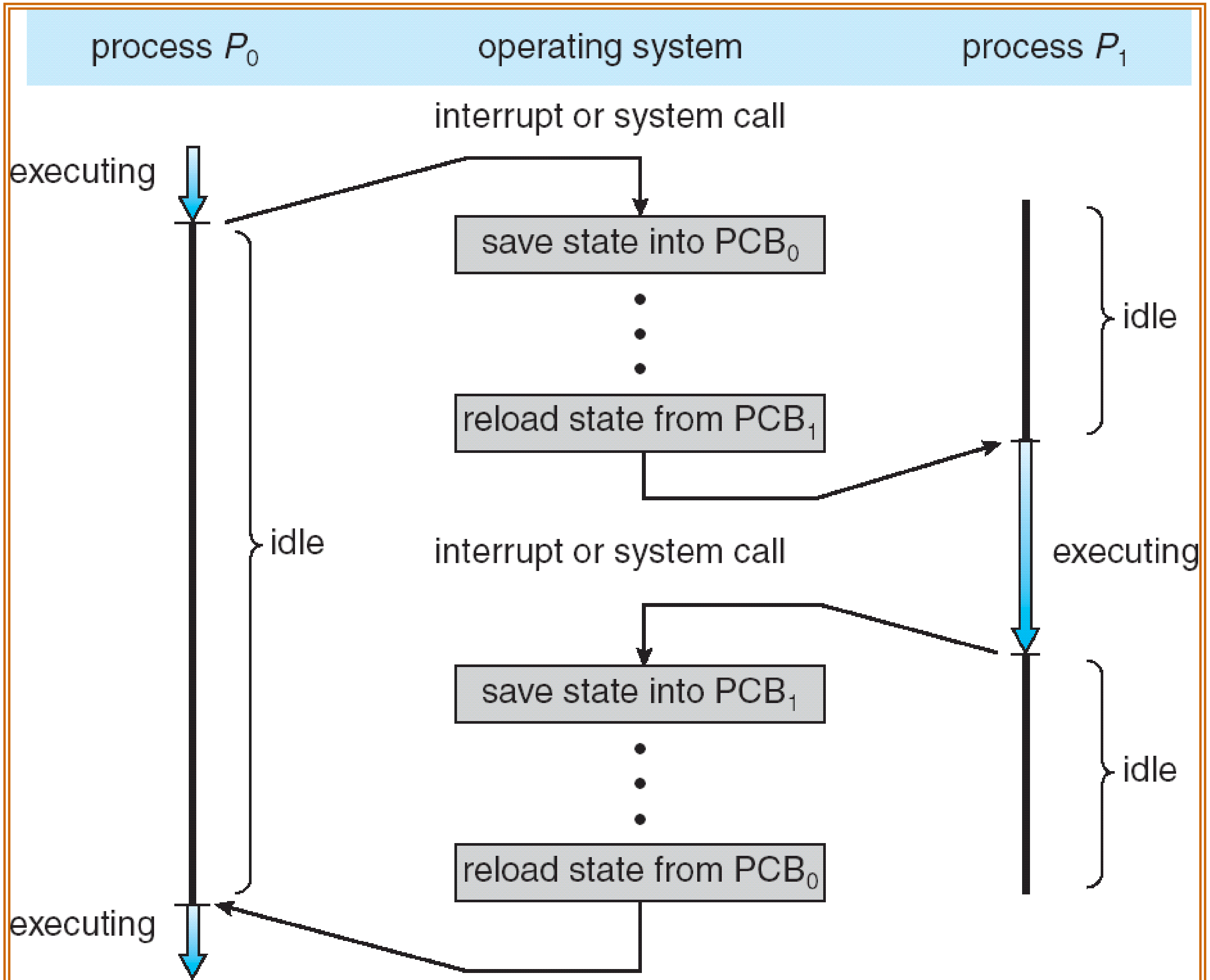


Context Switch

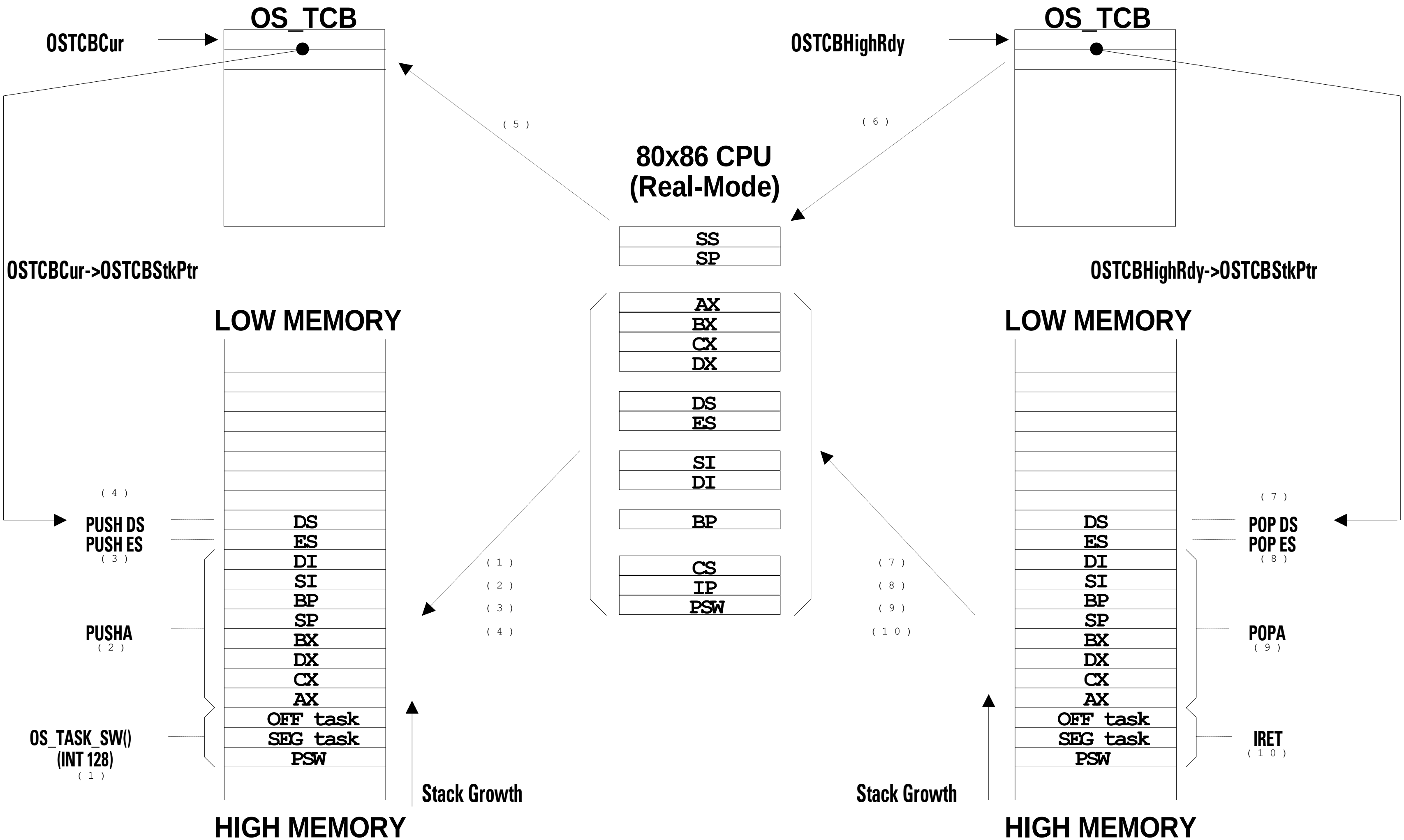
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is an overhead; the system does no useful work while switching
- Time dependent on hardware
    - Roughly 2000 ns/cxtsw on Intel 5150 (2.66 GHz)
    - And the subsequent costs of pipeline stall and cache pollution

http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

CPU Switch From Process to Process



Example: Context Switch in **uC/OS-2**



```
;*********************************************************************************************************
*
;                                PERFORM A CONTEXT SWITCH (From task level)
;                                           void OSCtxSw(void)
;
; Note(s): 1) Upon entry,
;             OSTCBCur     points to the OS_TCB of the task to suspend
;             OSTCBHighRdy points to the OS_TCB of the task to resume
;
;          2) The stack frame of the task to suspend looks as follows:
;
;                 SP -> OFFSET  of task to suspend    (Low memory)
;                       SEGMENT of task to suspend
;                       PSW     of task to suspend    (High memory)
;
;          3) The stack frame of the task to resume looks as follows:
;
;                 OSTCBHighRdy->OSTCBStkPtr --> DS                              (Low memory)
;                                               ES
;                                               DI
;                                               SI
;                                               BP
;                                               SP
;                                               BX
;                                               DX
;                                               CX
;                                               AX
;                                               OFFSET  of task code address
;                                               SEGMENT of task code address
;                                               Flags to load in PSW            (High memory)
;*********************************************************************************************************
*
```

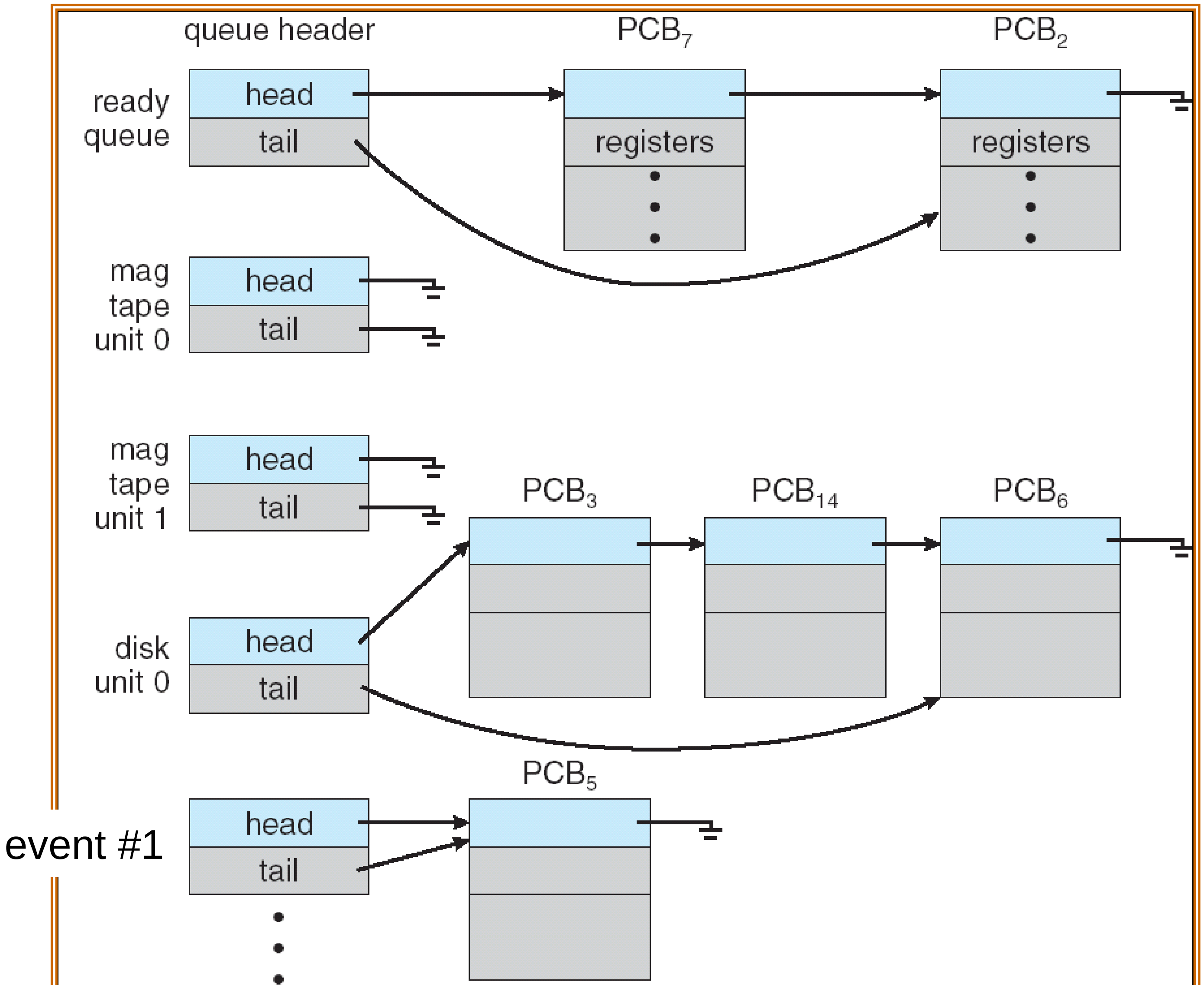
```
_OSCtxSw    PROC    FAR
;
            PUSHA                                  ; Save current task's context
            PUSH    ES                             ;
            PUSH    DS                             ;
;
            MOV     AX, SEG _OSTCBCur              ; Reload DS in case it was altered
            MOV     DS, AX                         ;
;
            LES     BX, DWORD PTR DS:_OSTCBCur     ; OSTCBCur->OSTCBStkPtr = SS:SP
            MOV     ES:[BX+2], SS                  ;
            MOV     ES:[BX+0], SP                  ;
;
            CALL    FAR PTR _OSTaskSwHook          ; Call user defined task switch hook
;
            MOV     AX, WORD PTR DS:_OSTCBHighRdy+2 ; OSTCBCur = OSTCBHighRdy
            MOV     DX, WORD PTR DS:_OSTCBHighRdy  ;
            MOV     WORD PTR DS:_OSTCBCur+2, AX    ;
            MOV     WORD PTR DS:_OSTCBCur, DX      ;
;
            MOV     AL, BYTE PTR DS:_OSPrioHighRdy ; OSPrioCur = OSPrioHighRdy
            MOV     BYTE PTR DS:_OSPrioCur, AL     ;
;
            LES     BX, DWORD PTR DS:_OSTCBHighRdy ; SS:SP = OSTCBHighRdy->OSTCBStkPtr
            MOV     SS, ES:[BX+2]                  ;
            MOV     SP, ES:[BX]                    ;
;
            POP     DS                             ; Load new task's context
            POP     ES                             ;
            POPA                                   ;
;
            IRET                                   ; Return to new task
;
_OSCtxSw    ENDP
```

PROCESS SCHEDULING

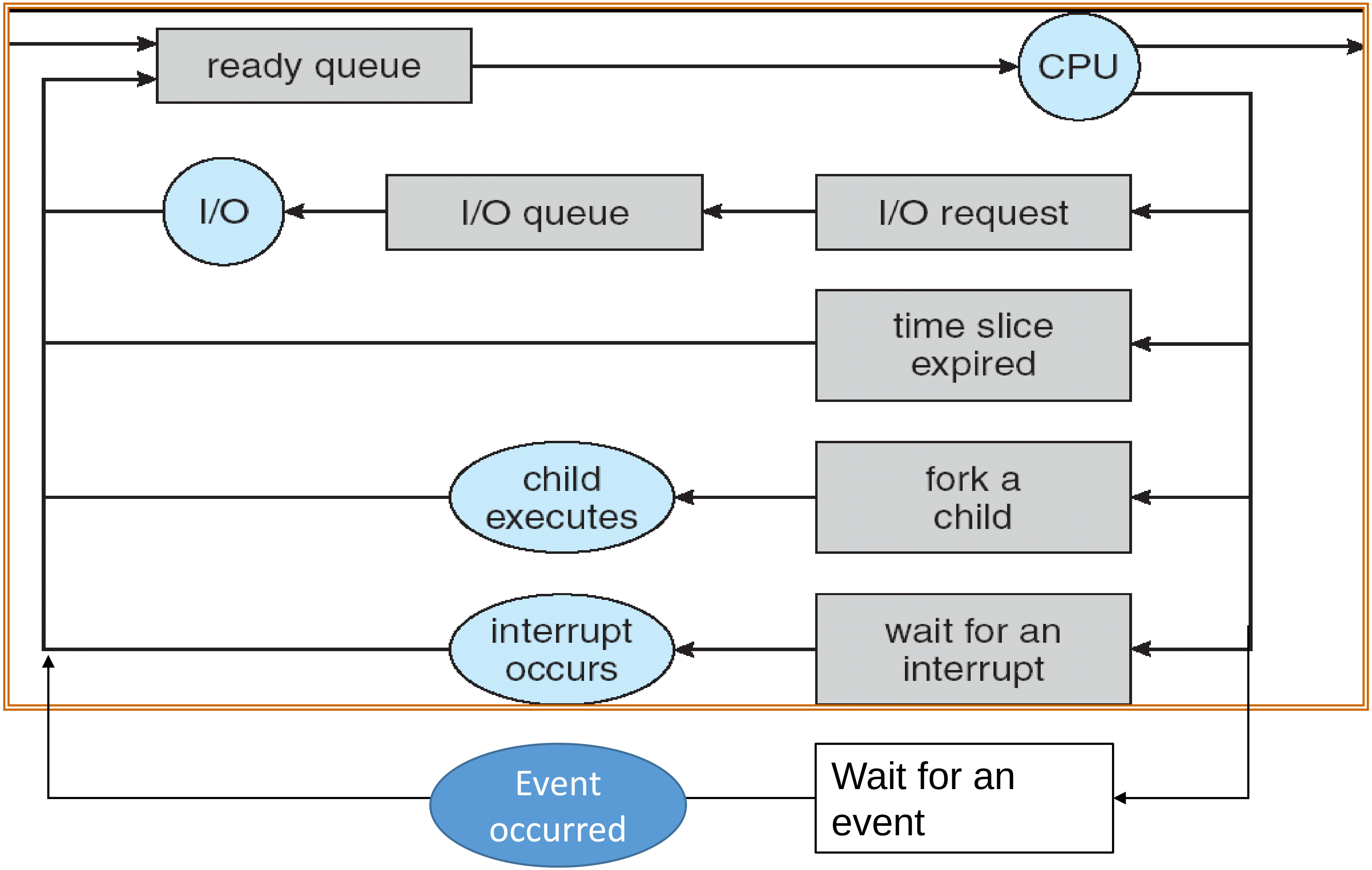
Process Scheduling Queues

- Ready queue – set of all processes residing in main memory, ready for execution
- Device queues – set of processes waiting for an I/O device
- Event queues – set of processes waiting for an event (e.g., semaphore)
- Processes migrate among the various queues

Various Process Queues



Representation of Process Scheduling



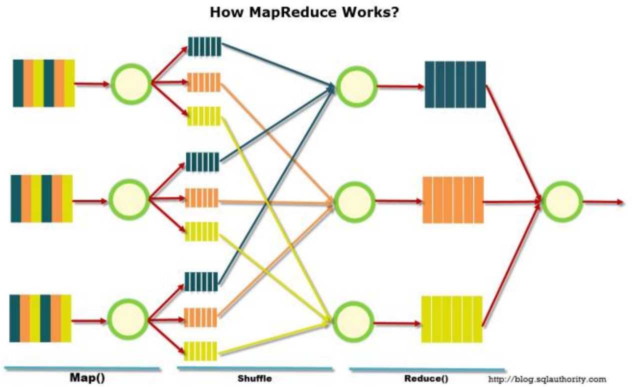
Schedulers

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast)

Schedulers (Cont.)

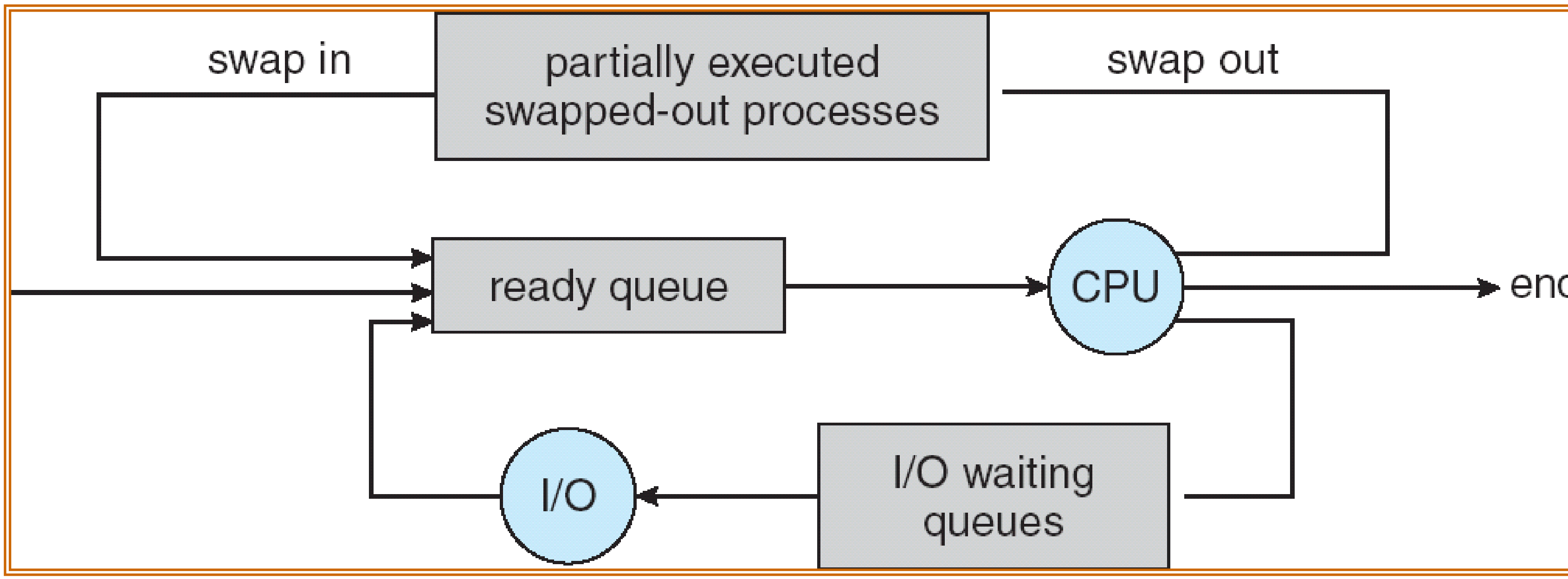
- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
- The long-term scheduler controls the degree of multiprogramming
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts

Long-Term Scheduler



- In batch-processing systems, the long-term scheduler is to make a good mix of I/O bound processes and CPU-bound processes
  - Modern batch processing example: MapReduce
- Timesharing systems do not have long-term schedulers
  - The user decides how many progrrams to be executed

Addition of Medium Term Scheduling



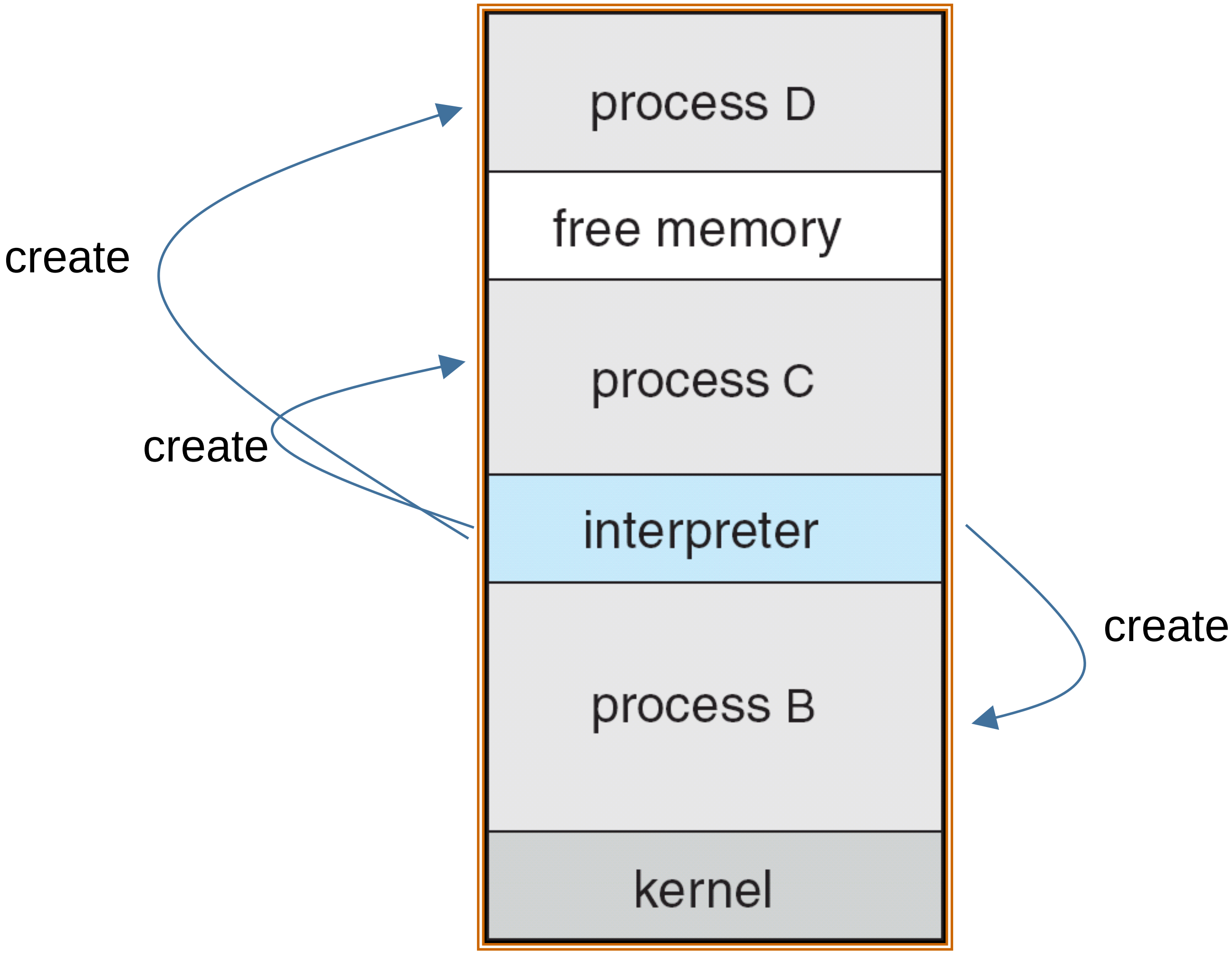
Swapping out: “saving” the memory image of a process (to a disk) to give memory space to new processes

Checklist

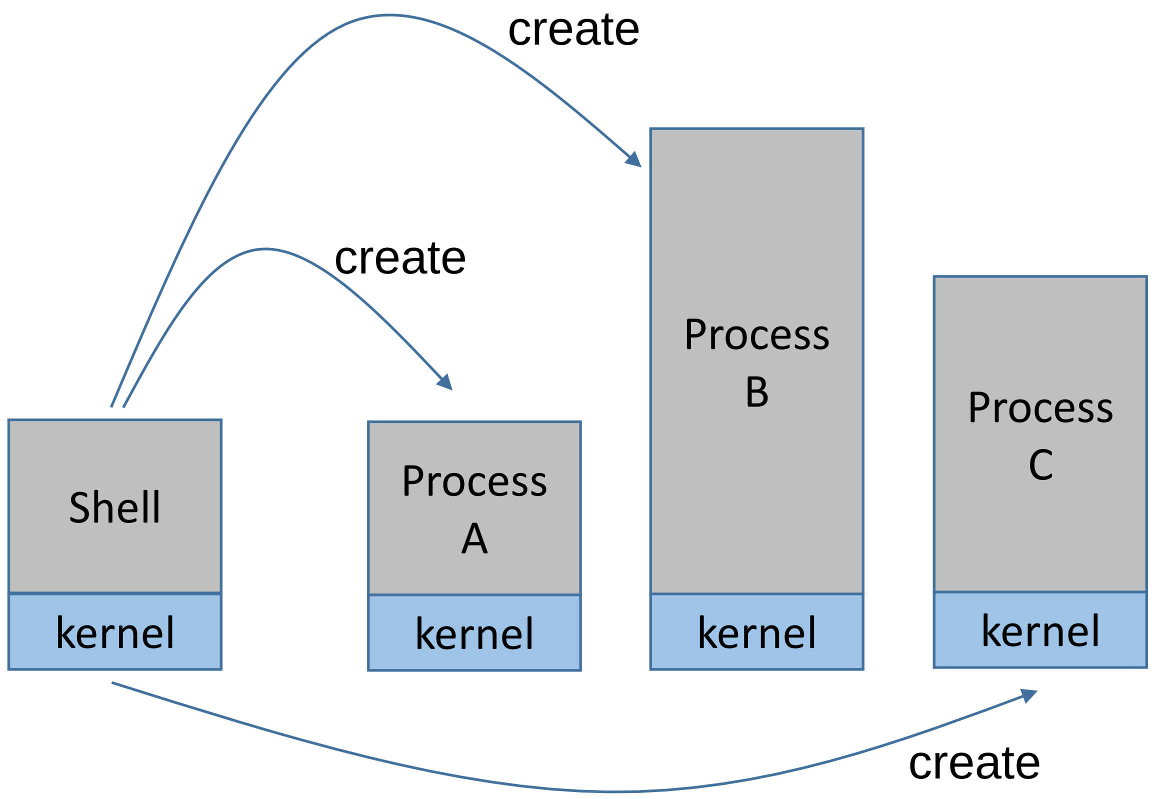
- Long term scheduler
- Short term scheduler
- Mid term scheduler
- I/O-bound and CPU-bound processes

OPERATIONS ON PROCESSES (CREATION & TERMINATION)

A Multiprogramming System (without Virtual Memory)



A Multiprogramming System (with Virtual Memory)



Processes cannot see each other as their memory spaces are completely separated

Process Creation

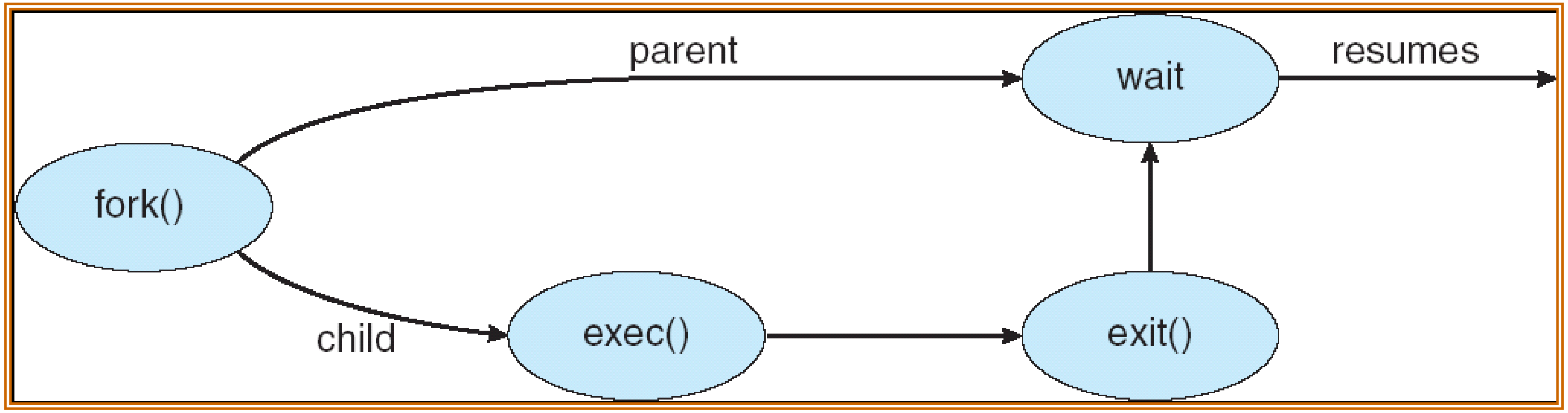
- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources,
  - children share subset of parent’s resources, or
  - parent and child share no resources
- Execution
  - Parent and children execute concurrently or
  - parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate its parent
  - Child has a program loaded into it
- UNIX examples
  - fork system call creates new process (clone the calling process)
  - exec system call used after a fork to replace the process' memory space with a new program

- Right after fork():
  - The child is **an exact copy** of the parent

Process Creation (UNIX)



C Program Forking Separate Process (UNIX)

Parent and child see different return values!!!

```
int main()
{
   pid_t  pid;
   /* fork another process */
   pid = fork();
   if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
   }
   else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
   }
}
```

The child won’t return here after exec()

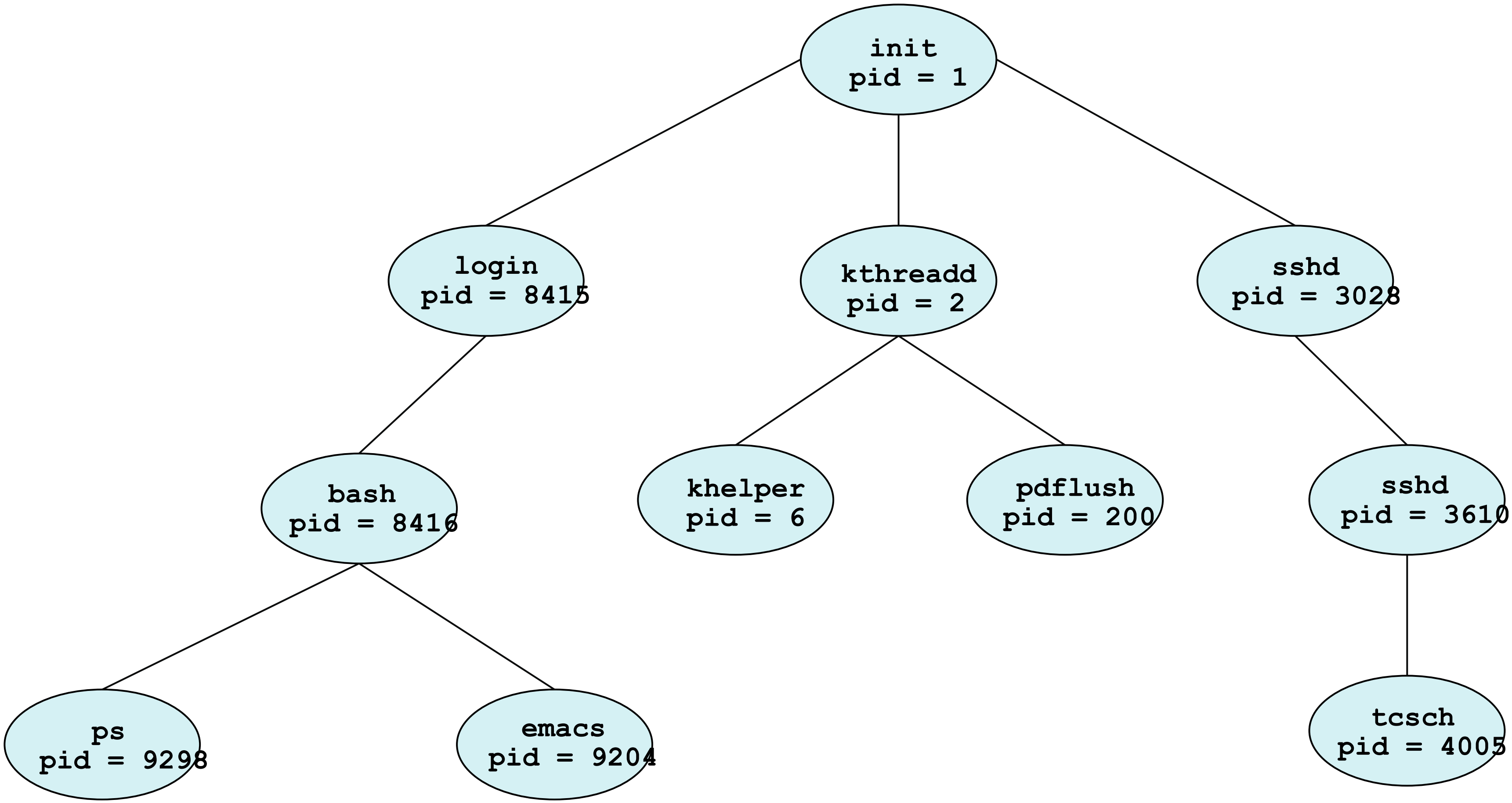
The parent has child’s pid so it can kill the child (if necessary)

Address Spaces of Parent and Child Processes

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int x=0;

int main()
{
        pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                x++;
                exit(0);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("%d",x);
                exit(0);
        }
}
```

What is the output of this program?

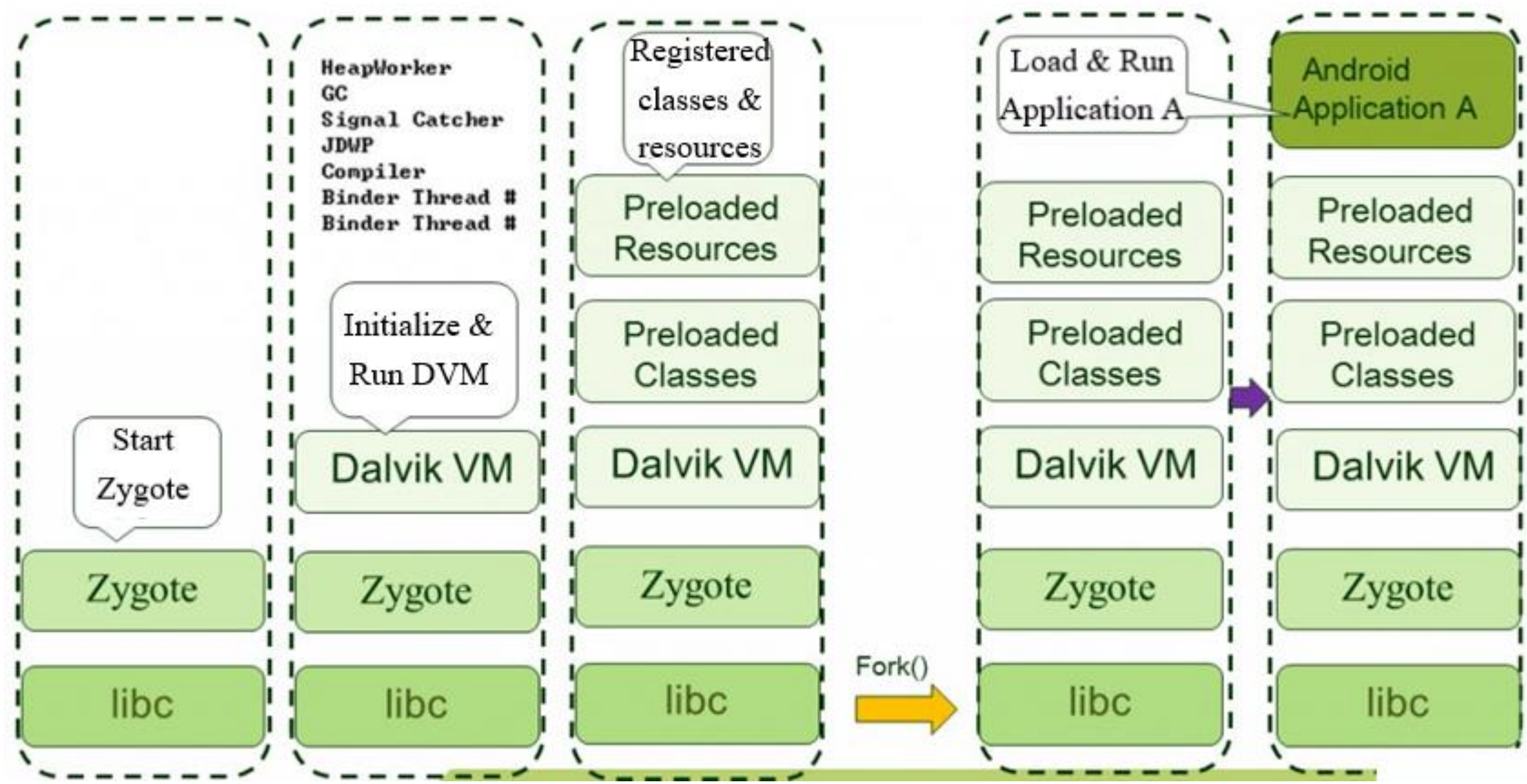
A Tree of Processes in Linux



The fork() → exec() Convention

- fork() requires to make a copy of the current process, but the following exec() replaces the address space
- The copying is efficiently implemented through memory mapping, with the assistance of the MMU hardware (see Virtual Memory)
- Use vfork() instead of fork() if the CPU is not equipped with an MMU

A Fair Use of fork() without exec()



https://www.itread01.com/p/1361603.html

vfork(): parent and child share most resources

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int x=0;

int main()
{
        pid_t  pid;
        /* fork another process */
        pid = vfork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                x++;     // often, here calls exec()
                _exit(0);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("%d",x);
                exit(0);
        }
}
```

What is the output of this program?

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - Browser process manages user interface, disk and network I/O
  - Renderer process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in sandbox restricting disk and network I/O, minimizing effect of security exploits
  - Plug-in process for each type of plug-in

Process Termination

- Process executes last statement and asks the operating system to delete itself
  - Calling the exit() system call
  - Synchronous termination
  - A return value must be retrieved by its parent (via wait())
- Parent may terminate execution of its children
  - Sending a signal (SIGKILL) to a child
  - Asynchronous termination

Orphan Processes and Zombie Processes

- A zombie (defunct) process
  - A process that has terminated (all resources released) but its return value has not been retrieved by its parent yet
  - It still occupies an entry of the process table
- An orphan process
  - A process whose parent process has terminated
  - Linux: an orphan will be adopted by process 0 (*init*), and *init* will wait/retrieve the return value of an orphan (note: implementation-dependent)

- Zombie implies orphan? Orphan implies zombie?

A Zombie Child Process

```
#include <stdio.h>
#include <sys/types.h>

main(){

        if(fork()==0){
                // child process
                printf("child pid=%d\n", getpid());
                exit(0)
        }

        // parent process
        sleep(20);    // let the child print the message
        printf("parent pid=%d \n", getpid());
        exit(0);
}
```

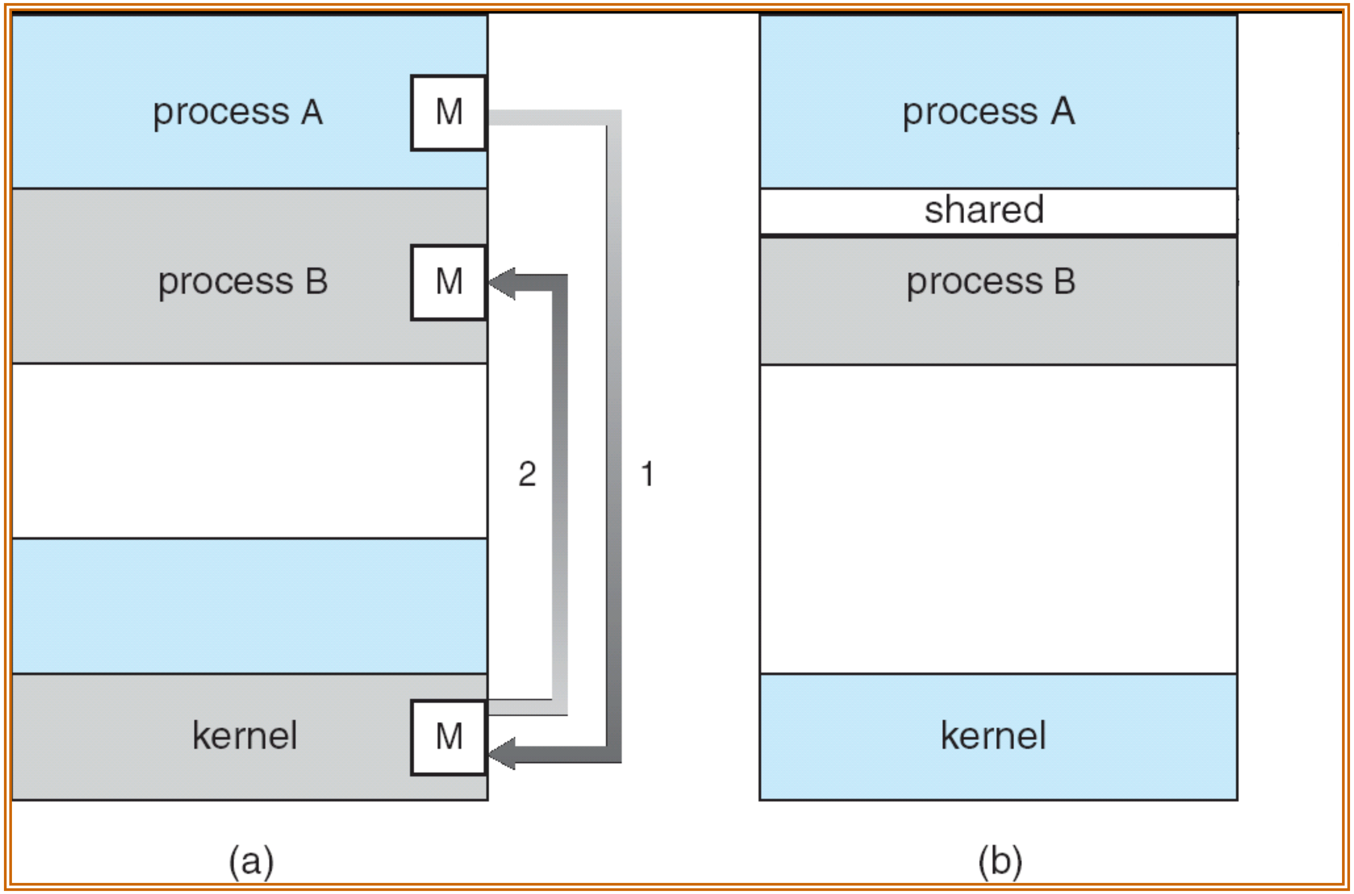
After the child terminates, it becomes a zombie until being adopted and handled by *init*.

INTER-PROCESS COMMUNICATION

Cooperating Processes

- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
  - UX improvement

Communications Models



Message passing **Shared memory**

IPC- SHARED MEMORY

Shared Memory

- Linux offers the following system calls for shared memory management
  - shmget() – create a block of shared memory
  - shmat() – attach shared memory to the current process's address space
  - shmdt() – detach shared memory from the current process's address space
  - shmctl() – control shared memory (including delete)
- Let us assume that a piece of shared memory has been setup between two processes

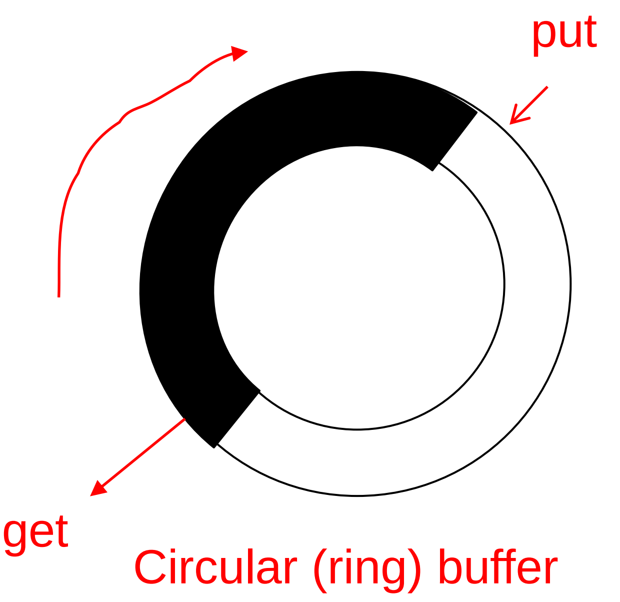
Producer-Consumer Problem

- Paradigm for cooperating processes, a producer process produces information that is consumed by a consumer process
  - The two processes run concurrently
- Objective:
  - to synchronize a producer and a consumer via shared memory
- Issues:
  - The buffer size is limited
  - Overwriting and null reading are not allowed

Bounded-Buffer – 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
- What are the conditions for buffer full and buffer empty?

Bounded-Buffer – Insert() Method

```
while (true) {

     /* Produce an item */
     while (( (in + 1) % BUFFER SIZE count)  == out)
          ;   /* do nothing -- no free buffers */

     buffer[in] = item;
     in = (in + 1) % BUFFER SIZE;
}
```

Bounded Buffer – Remove() Method

```
while (true) {

     while (in == out)
        ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
     return item;
}
```

Bounded Buffer Problem

- Why not to use a free-slot counter?
- Does this approach efficiently utilize CPU cycles?

- Generally, if two processes exchange data through shared memory, they require proper synchronization (see Synchronization)

IPC- MESSAGE PASSING

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message)
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

Interprocess Communication (IPC)

- Messages can be buffered in the link
  - P→[link <buffer>]→Q

- P will be *blocked* on sending if the link buffer is full
- Q will be *blocked* on receiving if the link buffer is empty
- Built-in synchronization between processes

Example: Linux Pipe

- A basic mechanism for IPC
  - For example: “ls | more”
  - A process “ls”, a process “more”, and a pipe between them
- The system call pipe() creates a pipe
  - Receiver must close the output side, and receives from the input side
  - Sender must close the input side, and write to the output side
  - A pipe is created and configured by the parent process

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);     // create the pipe before calling fork()

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
```

```
	if(childpid == 0)
	{
		/* Child process closes up input side of pipe */
		close(fd[0]);

		/* Send "string" through the output side of pipe */
		write(fd[1], string, (strlen(string)+1));
		exit(0);
	}
	else
	{
		/* Parent process closes up output side of pipe */
		close(fd[1]);

		/* Read in a string from the pipe */
		nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
		printf("Received string: %s", readbuffer);
	}

	return(0);
}
```

UNIX Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled
- Analogy
  - Interrupts for CPU (async or sync)
  - Signals for processes (async or sync)

Signal Handling

- Synchronous signals
  - A signal that is delivered to the process caused the event
  - E.g., divide overflow and memory-access violations
- Asynchronous signals
  - A signal that is delivered to a process other than the signaling process
  - E.g., the kill signal
- Signal handlers
  - Default handlers
  - User-defined handlers (using signal() or sigaction())

UNIX Signal Example

- Synchronous signals
  - SIGSEGV : Memory protection fault
  - SIGFPE : Arithmetic fault, including divided by zero
- Asynchronous signals
  - SIGKILL : Kill a process → cannot be captured :)
  - SIGSTOP : Suspend a process
  - SIGCHLD: A child terminates

Handling SIGSEGV on your own

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void sigsegv_handler(int sig) {
        printf("Received segmentation violation (SIGSEGV). \n");
        exit(0);
}

int main() {
        int *null_pointer=(int *)NULL;
        signal(SIGSEGV,sigsegv_handler);

        printf("About to segfault:\n");
        *null_pointer=0;

        printf("Shouldn't be here!\n");
        return 1;
}
```

Handling SIGSEGV on your own

```
void action(int sig, siginfo_t* siginfo, void* context)
{
    sig=sig; siginfo=siginfo;

    // get execution context
    mcontext_t* mcontext = &((ucontext_t*)context)->uc_mcontext;

    uint8_t* code = (uint8_t*)mcontext->gregs[REG_EIP];
    if (code[0] == 0x88 && code[1] == 0x10) { // mov %dl,(%eax)
        mcontext->gregs[REG_EIP] += 2; // skip it!
        return;
    }
}
main()
{
    ...
    sigaction(SIGSEGV, ...);
    ...
    for (int i = 0; i < 10; i++) { ((unsigned char*)0)[i] = i; }
}
```


http://stackoverflow.com/questions/6981702/how-to-write-a-segmentation-fault-handler-so-that-the-faulty-instruction-is-not

End of Chapter 3

Review Questions

1. Discuss which memory section that local variables and global variables are allocated from
2. Why a process transits from running to ready?
3. Discuss the details of a full context switch
4. Why fork() is slow without hardware support?
5. A piece of shared memory cannot be referenced without a shmat() call. Why?
6. What does pipe(), dup(), and dup2() do?
7. Discuss the pros and cons of shared memory and message passing
8. What are orphans and zombies? How can they be handled?
9. How do you write a program to roughly measure the context switch overhead in UNIX?

Review Questions

- *iowait* (p) is the time proportion that a process waits on I/O completion. Try to interpret the results below