CS222
Operating Systems

Lecture 05
Process Concept

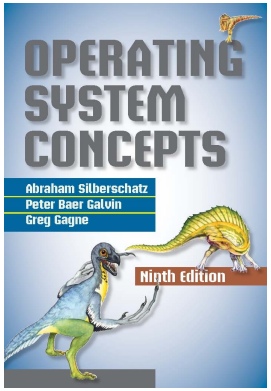
(Section 100001)

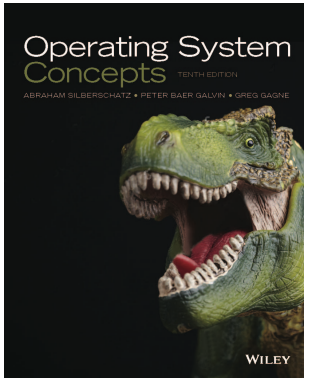
ผศ. ดร. กษิดิศ ชาญเชี่ยว

ckasidit@tu.ac.th

Textbook

- Avi Silberschatz, Peter B. Galvin and Greg Gagne; Operating System Concepts, 9th Edition; John Wiley & Sons, Inc; 2012; ISBN 978-1118063330

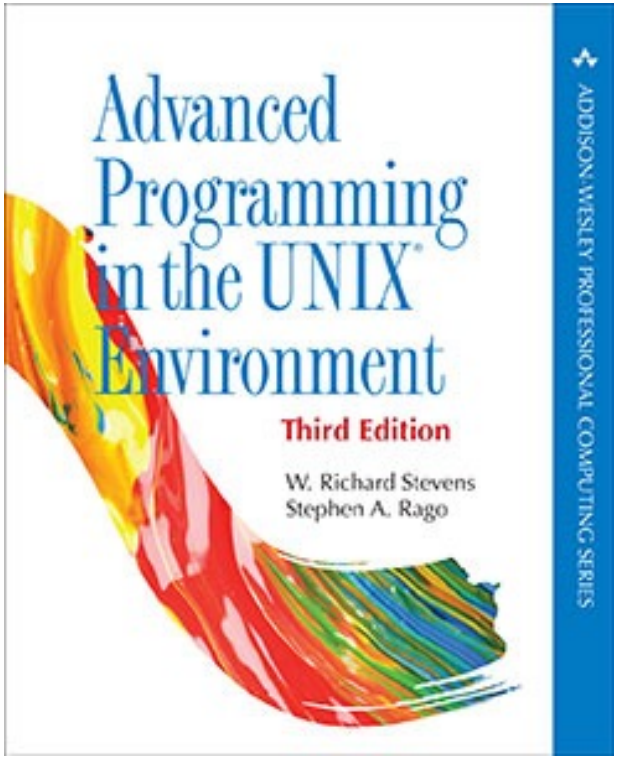




- Original Slides
- https://www.os-book.com/OS9/slide-dir/index.html
- https://www.os-book.com/OS10/slide-dir/index.html

Textbook

- Advanced Programming in the UNIX Environments
- http://www.apuebook.com/



Chapter 3 (Part 1)

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system.
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.

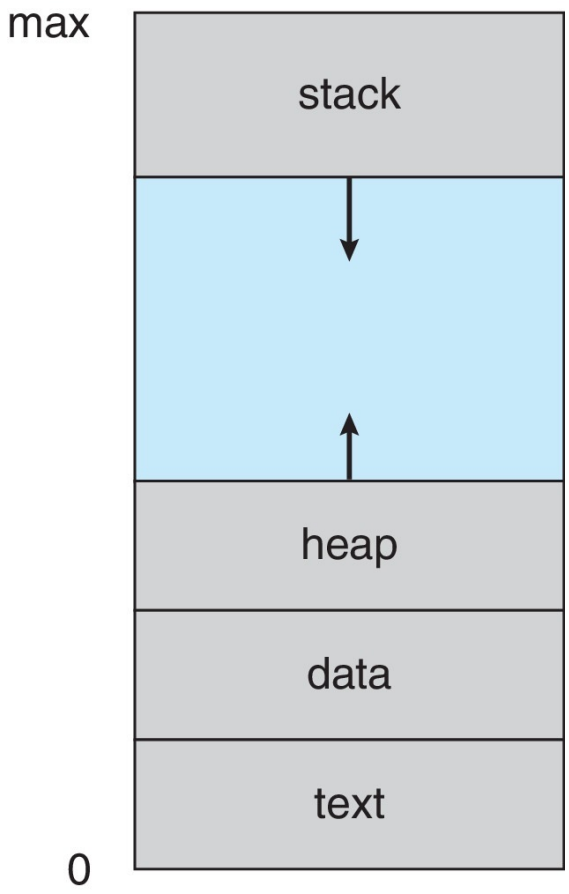
Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

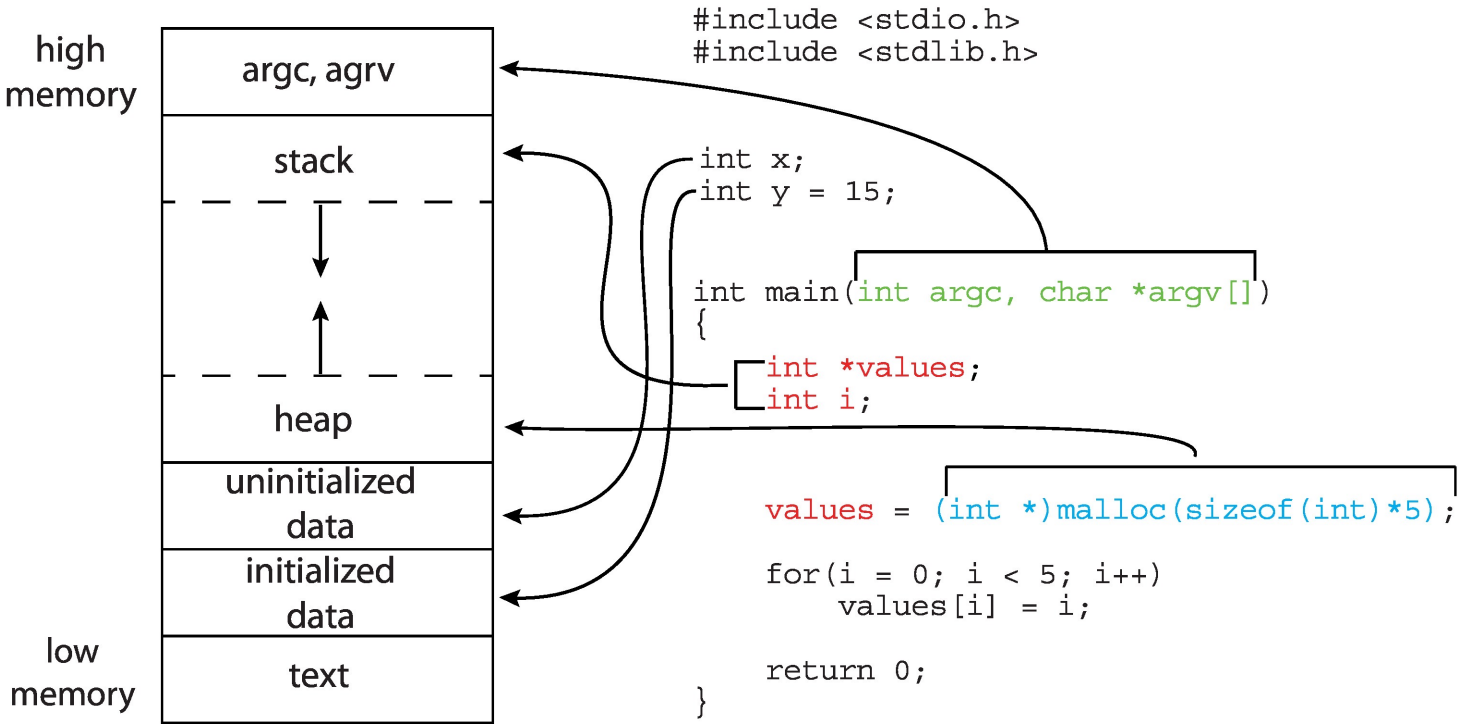
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**); process is **active**
  - Program becomes process when an executable file is loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - Consider multiple users executing the same program

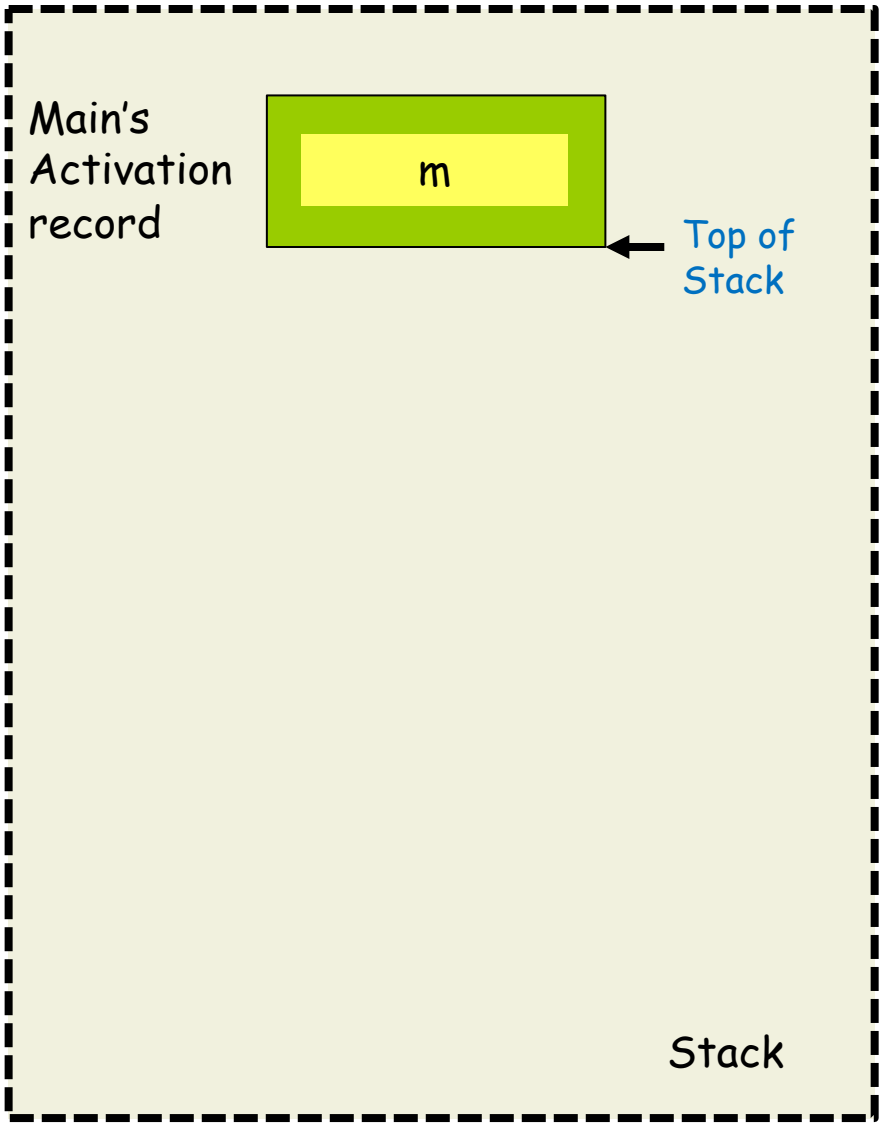
Process in Memory



Memory Layout of a C Program

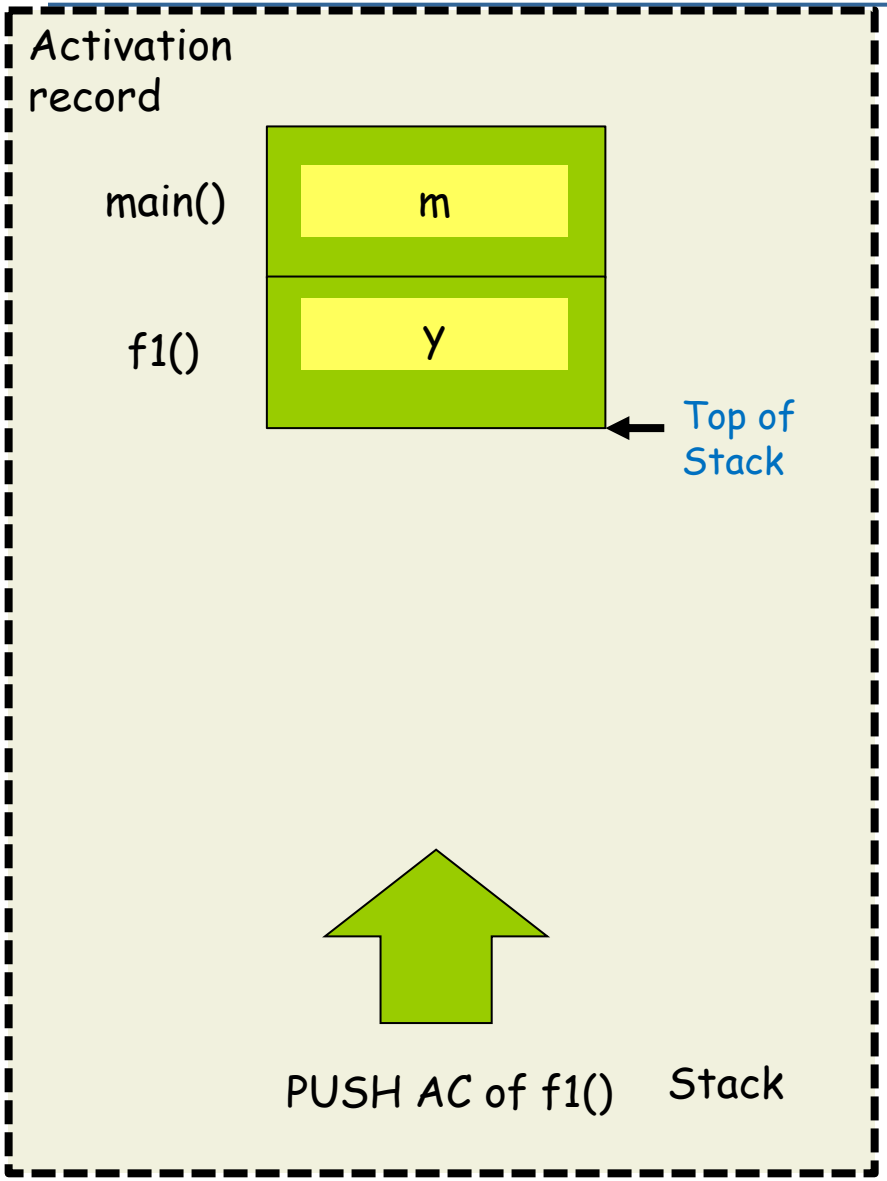


Stack



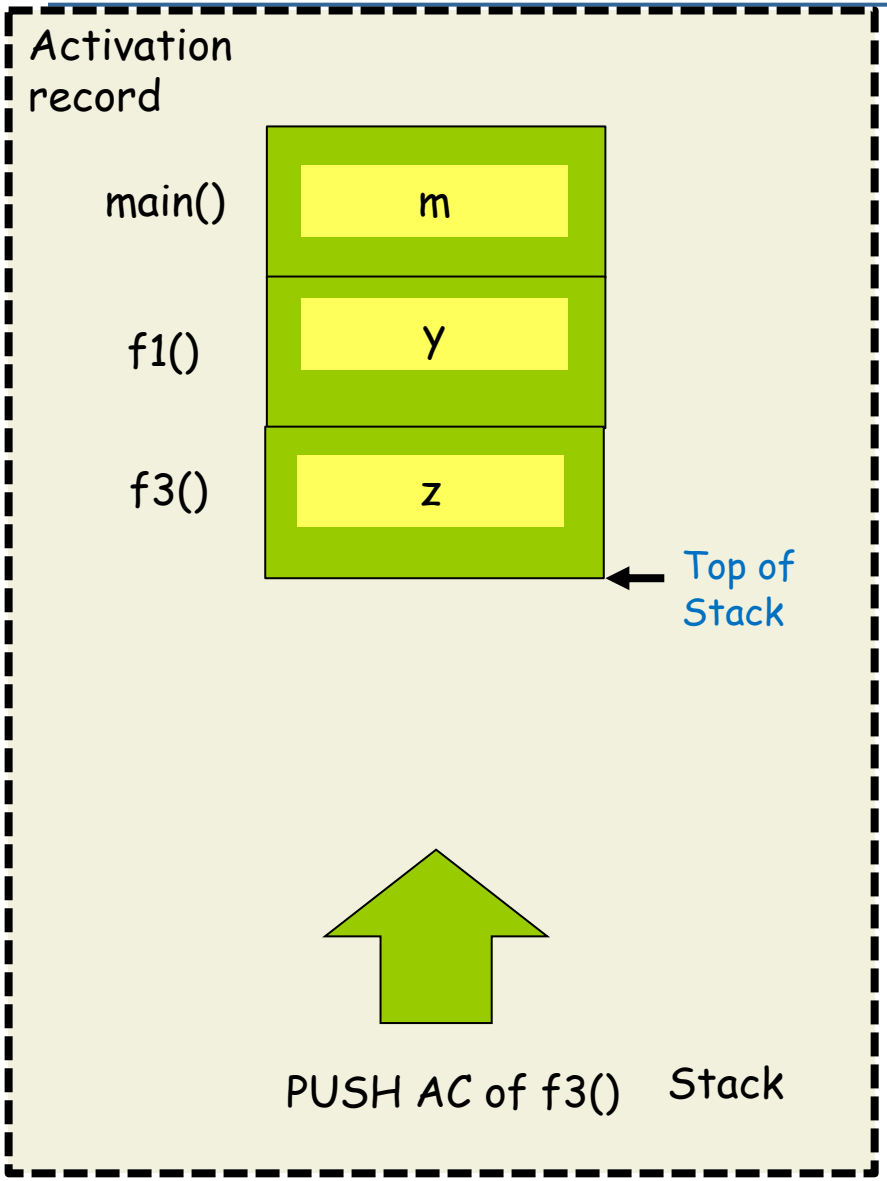

```
main(){
        int m
        f1();
        f2();
}
f1(){
        int x
        f3();
}
f2(){
        int y
        f3();
}
f3(){
        int y
        printf("hello");
}
```

Stack



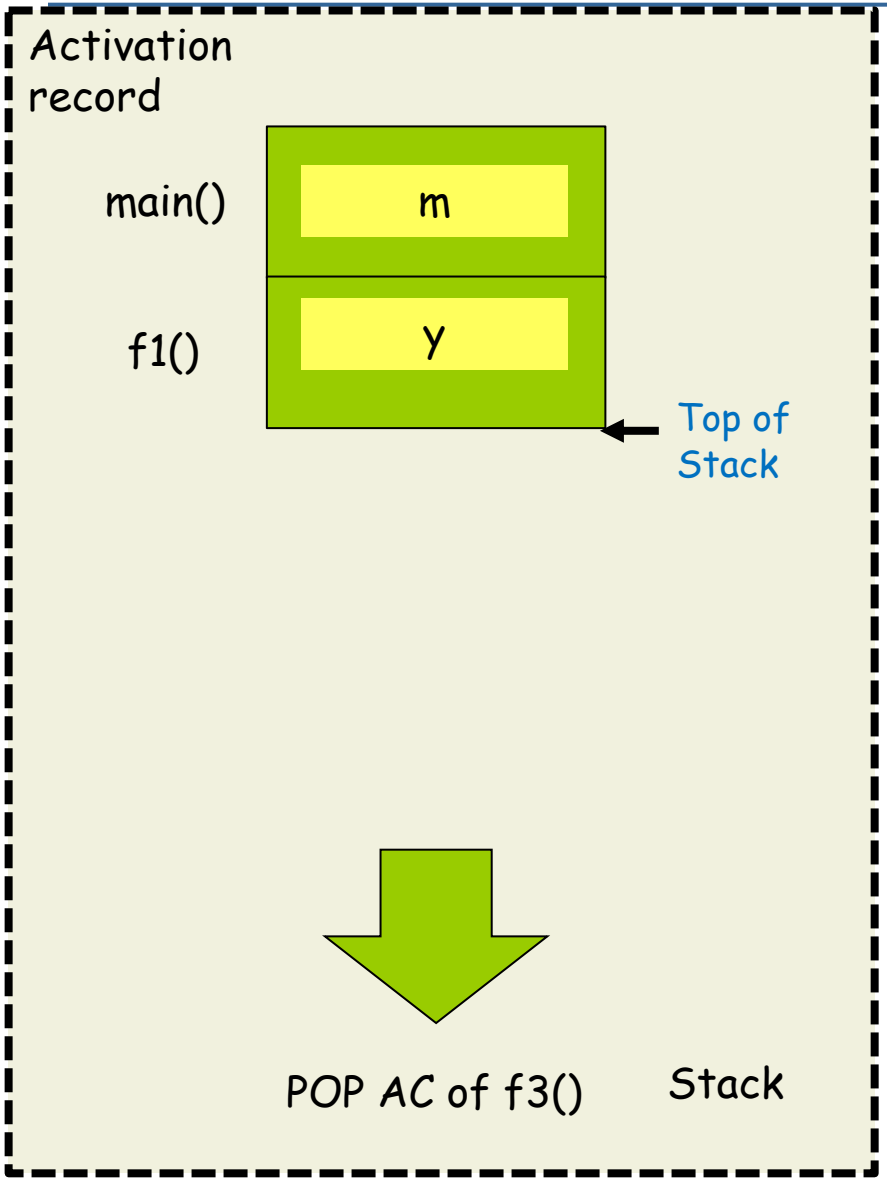

```
main(){
        int m
        f1();
        f2();
}
f1(){
        int x
        f3();
}
f2(){
        int y
        f3();
}
f3(){
        int y
        printf("hello");
}
```

Stack



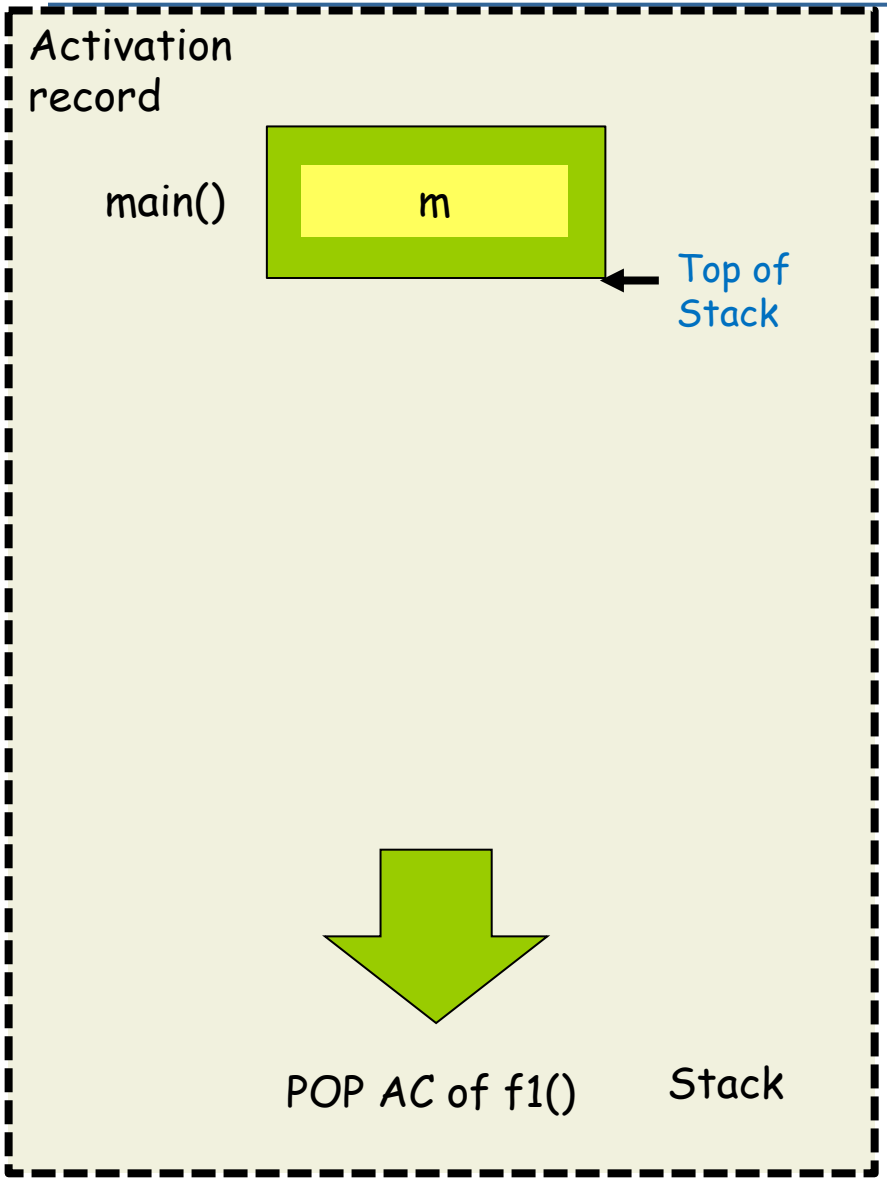

```
main(){
        int m
        f1();
        f2();
}
f1(){
        int x
        f3();
}
f2(){
        int y
        f3();
}
f3(){
        int z
        printf("hello");
}
```

Stack



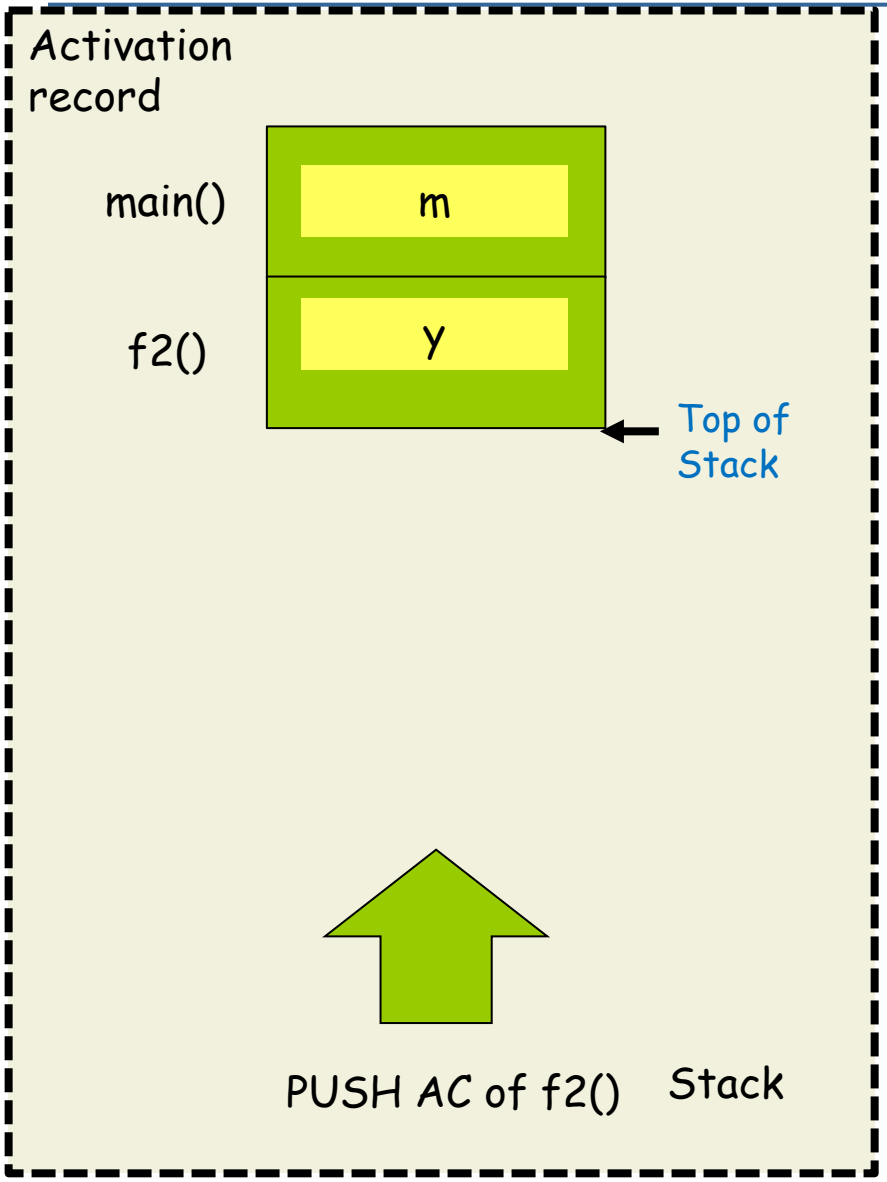

```
main(){
        int m
        f1();
        f2();
}
f1(){
        int x
        f3();
}
f2(){
        int y
        f3();
}
f3(){
        int z
        printf("hello");
}
```

Stack

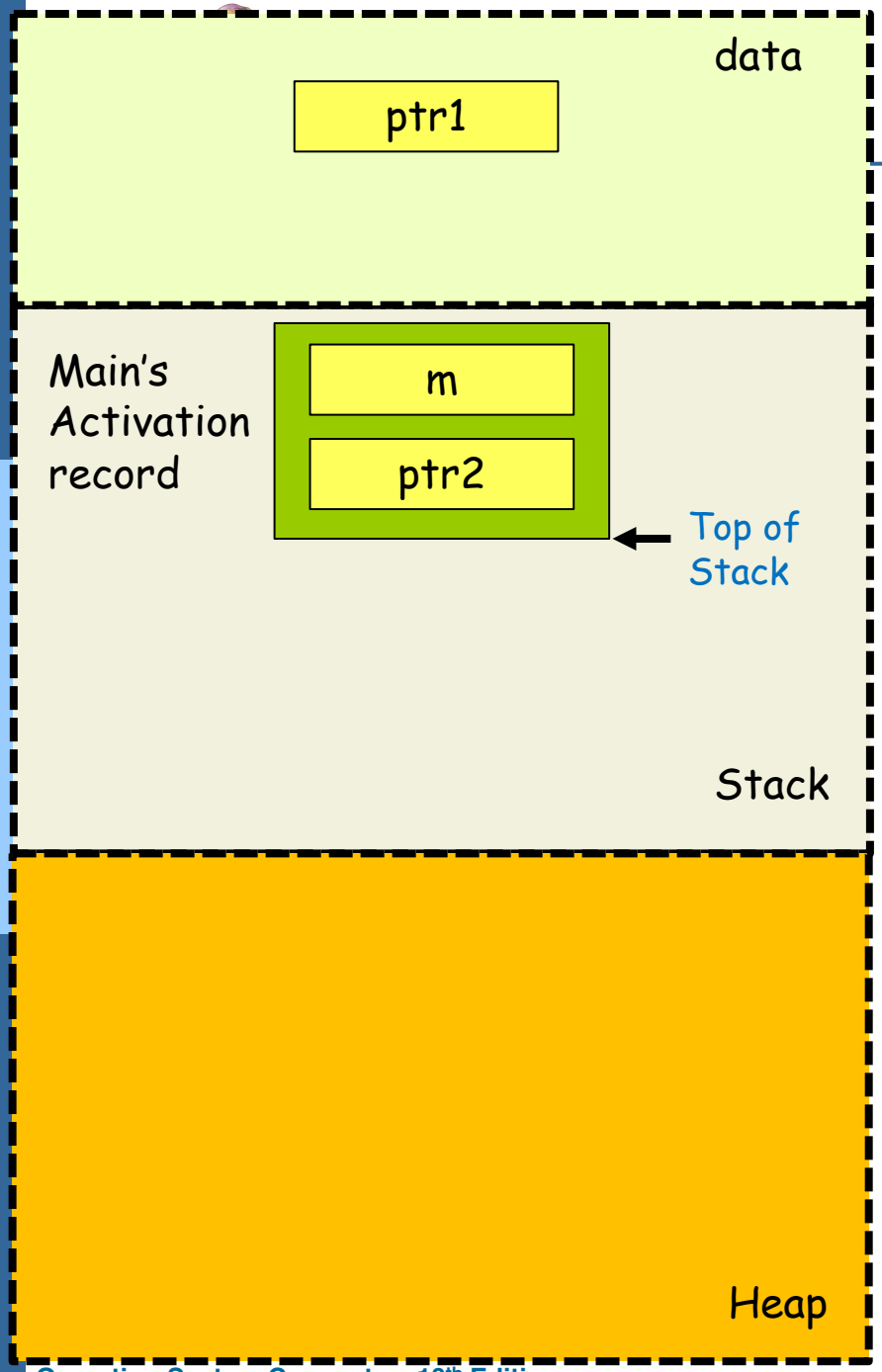



```
main(){
        int m
        f1();
        f2();
}
f1(){
        int x
        f3();
}
f2(){
        int y
        f3();
}
f3(){
        int z
        printf("hello");
}
```

Stack

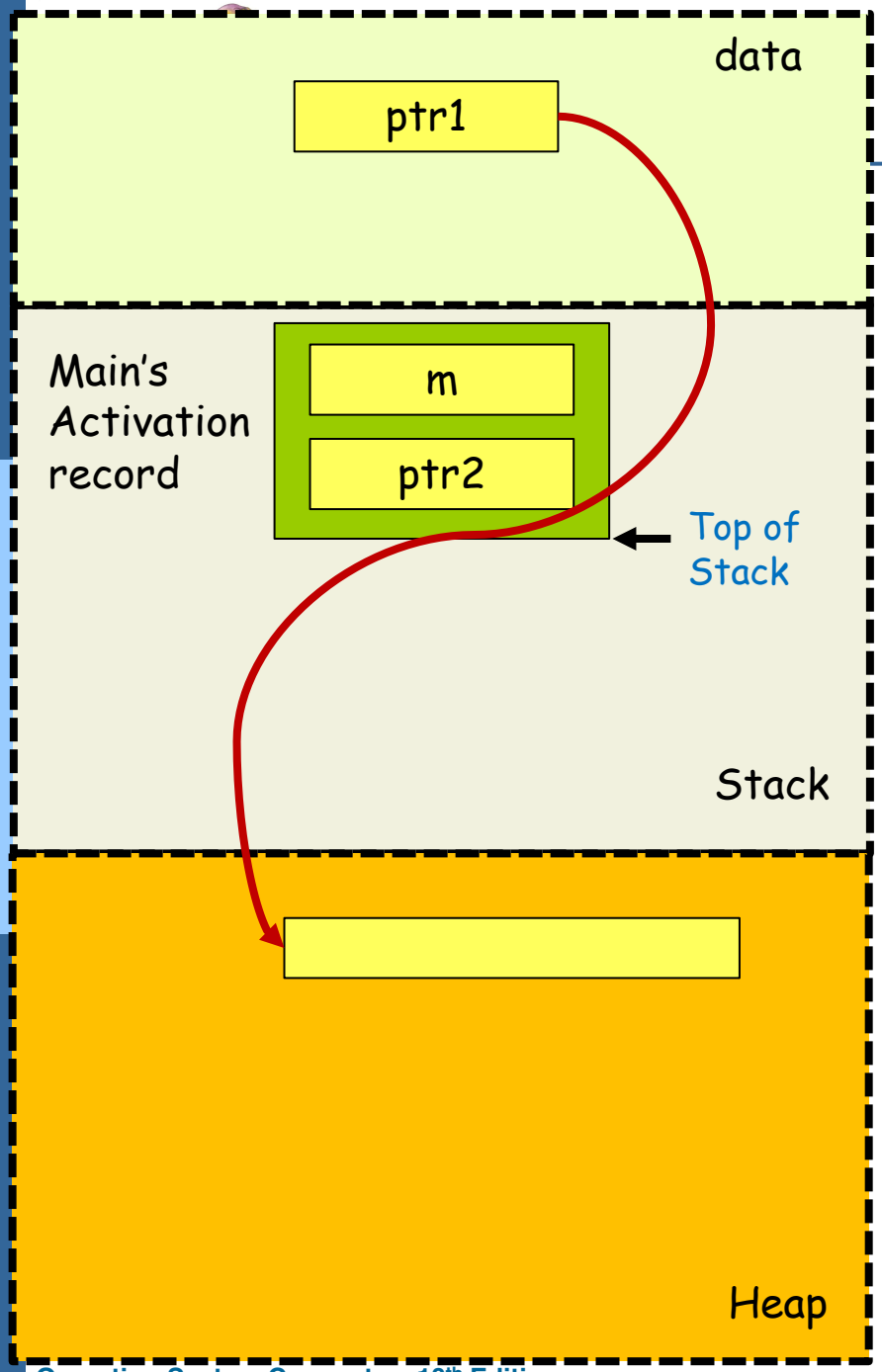



```
main(){
        int m
        f1();
        f2();
}
f1(){
        int x
        f3();
}
f2(){
        int y
        f3();
}
f3(){
        int y
        printf("hello");
}
```



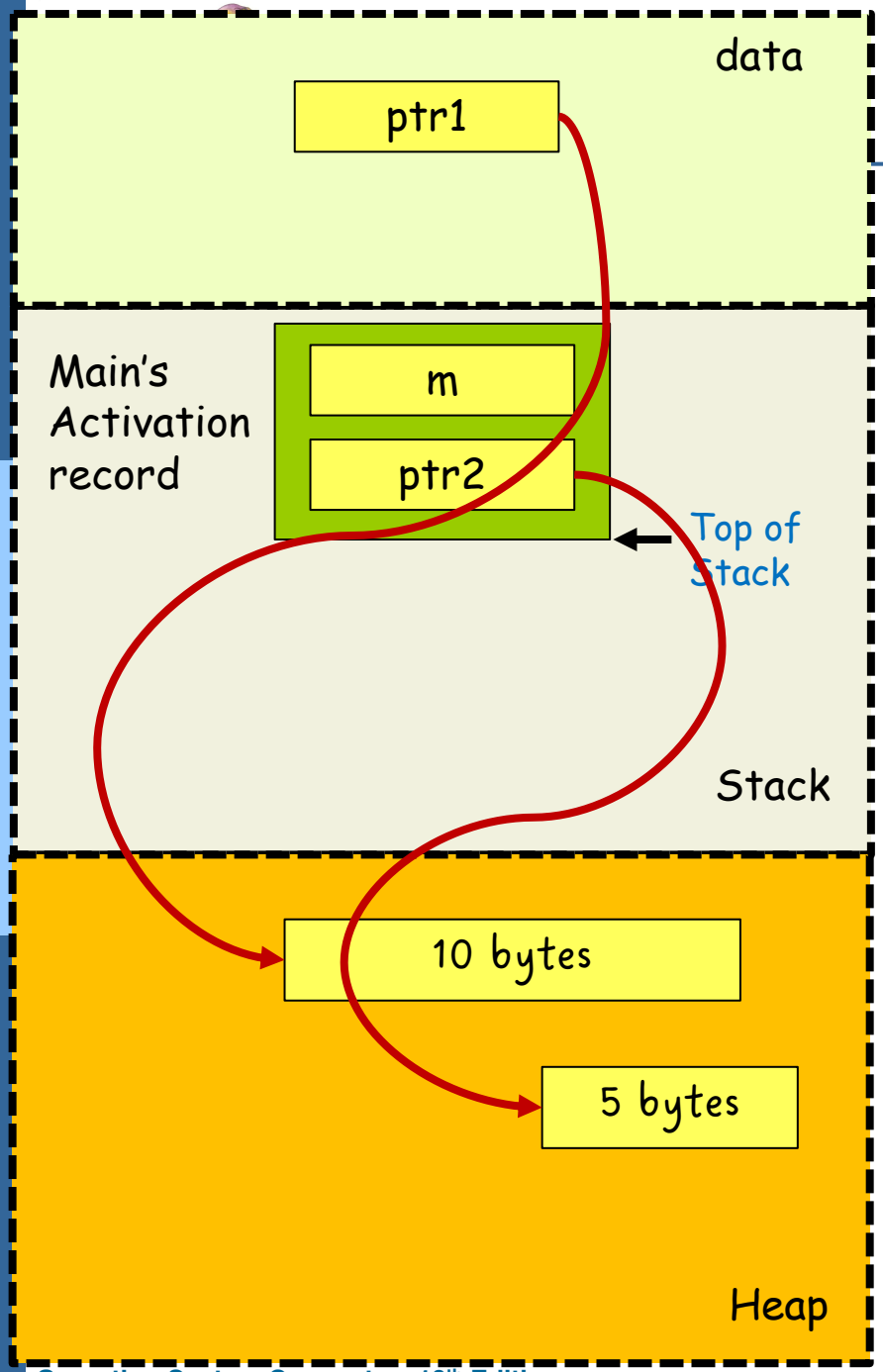
Heap

```
char *ptr1
main(){
        int m
        char *ptr2

        ptr1 = malloc(10)
        ptr2 = malloc(5)
        f1();

}

f1(){
        int x
        char ptr3

        ptr3 = malloc(5)
        printf("hello world")
        free(ptr3)
}
```



Heap

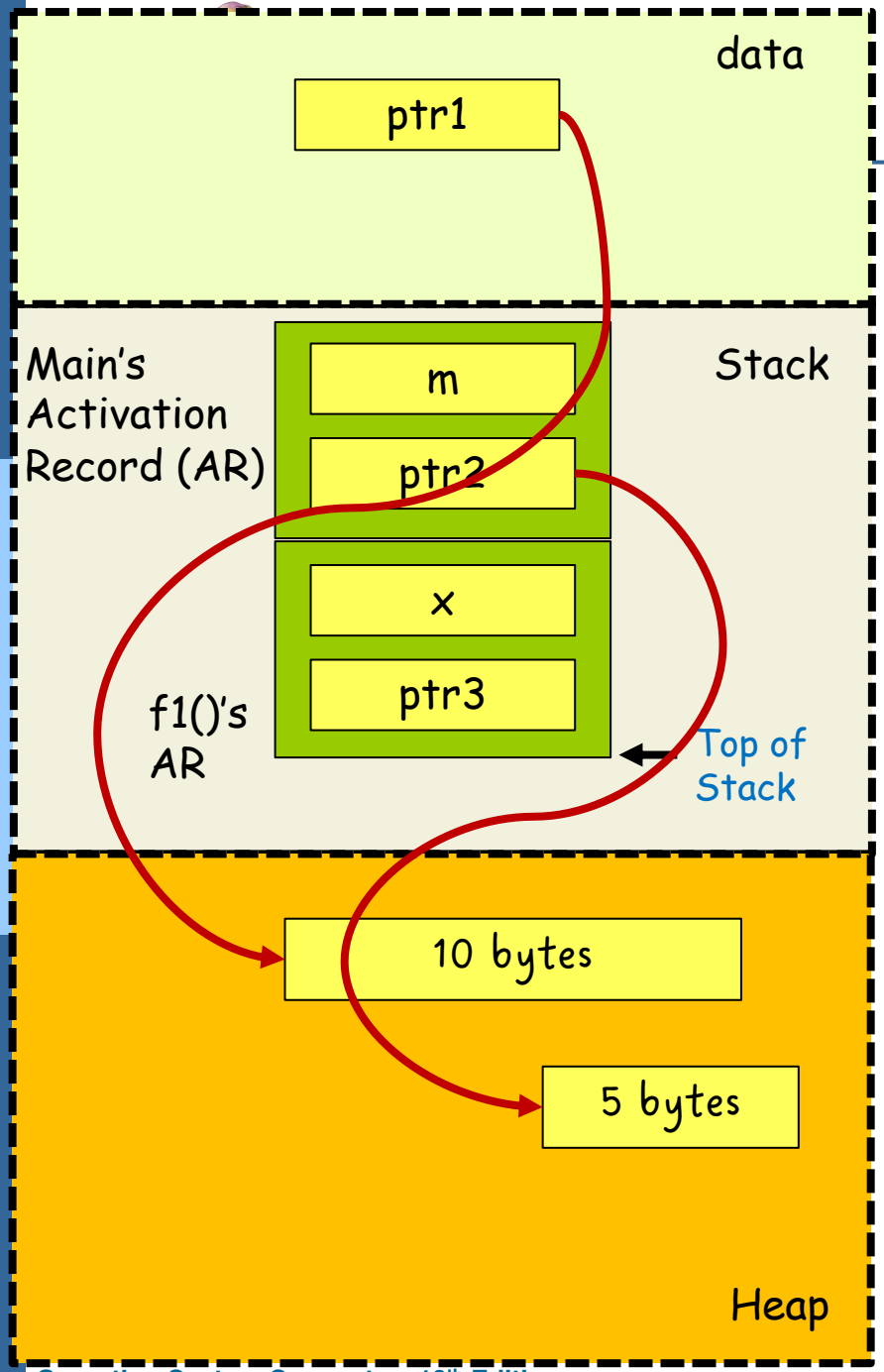
```
char * ptr1
main(){
      int m
      char * ptr2

      ptr1 = malloc(10)
      ptr2 = malloc(5)
      f1();

}

f1(){
      int x
      char * ptr3

      ptr3 = malloc(5)
      printf("hello world")
      free(ptr3)
}
```



Heap

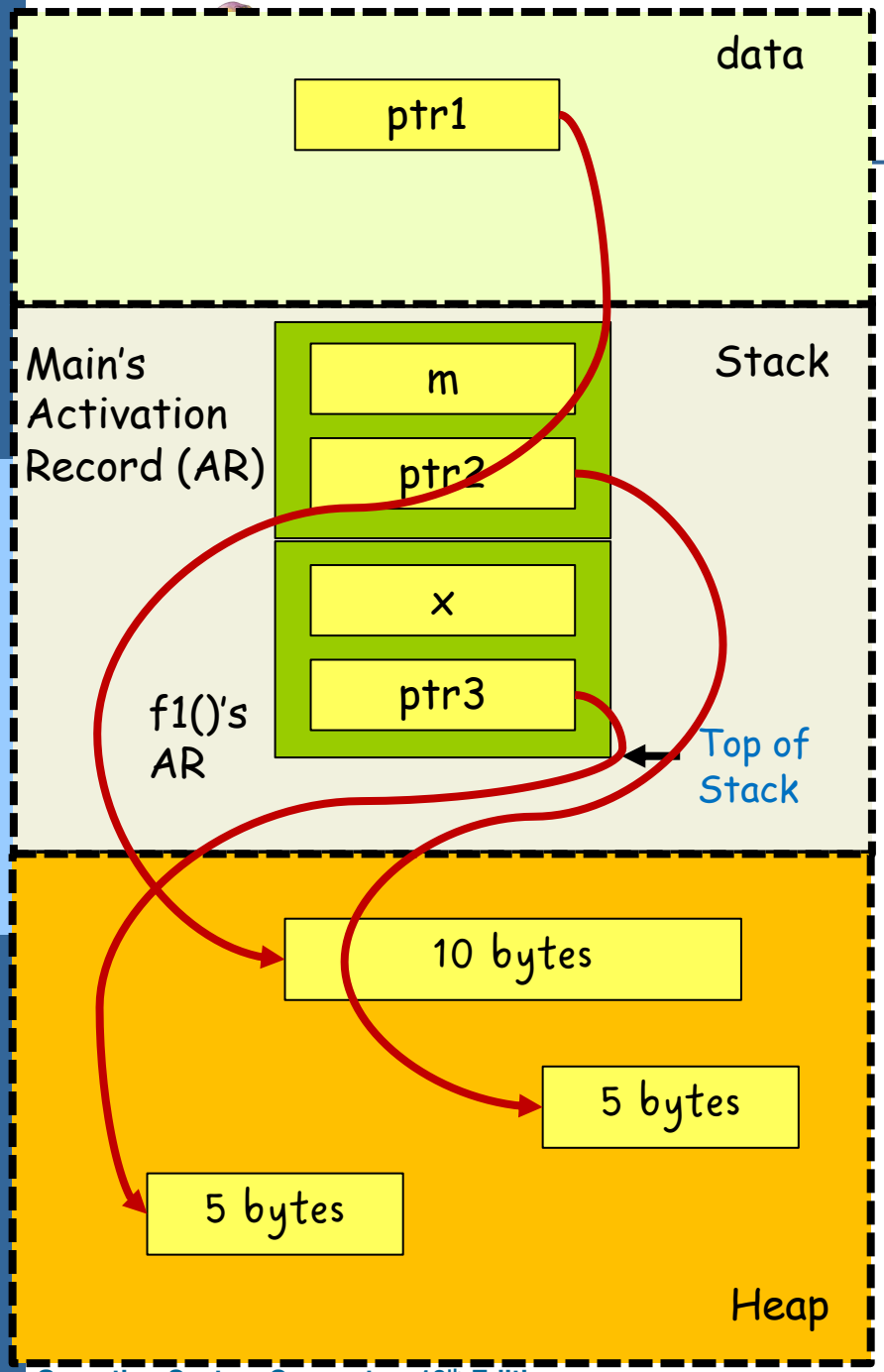
```
char * ptr1
main(){
      int m
      char * ptr2

      ptr1 = malloc(10)
      ptr2 = malloc(5)
      f1();

}

f1(){
      int x
      char * ptr3

      ptr3 = malloc(5)
      printf("hello world")
      free(ptr3)
}
```

Heap

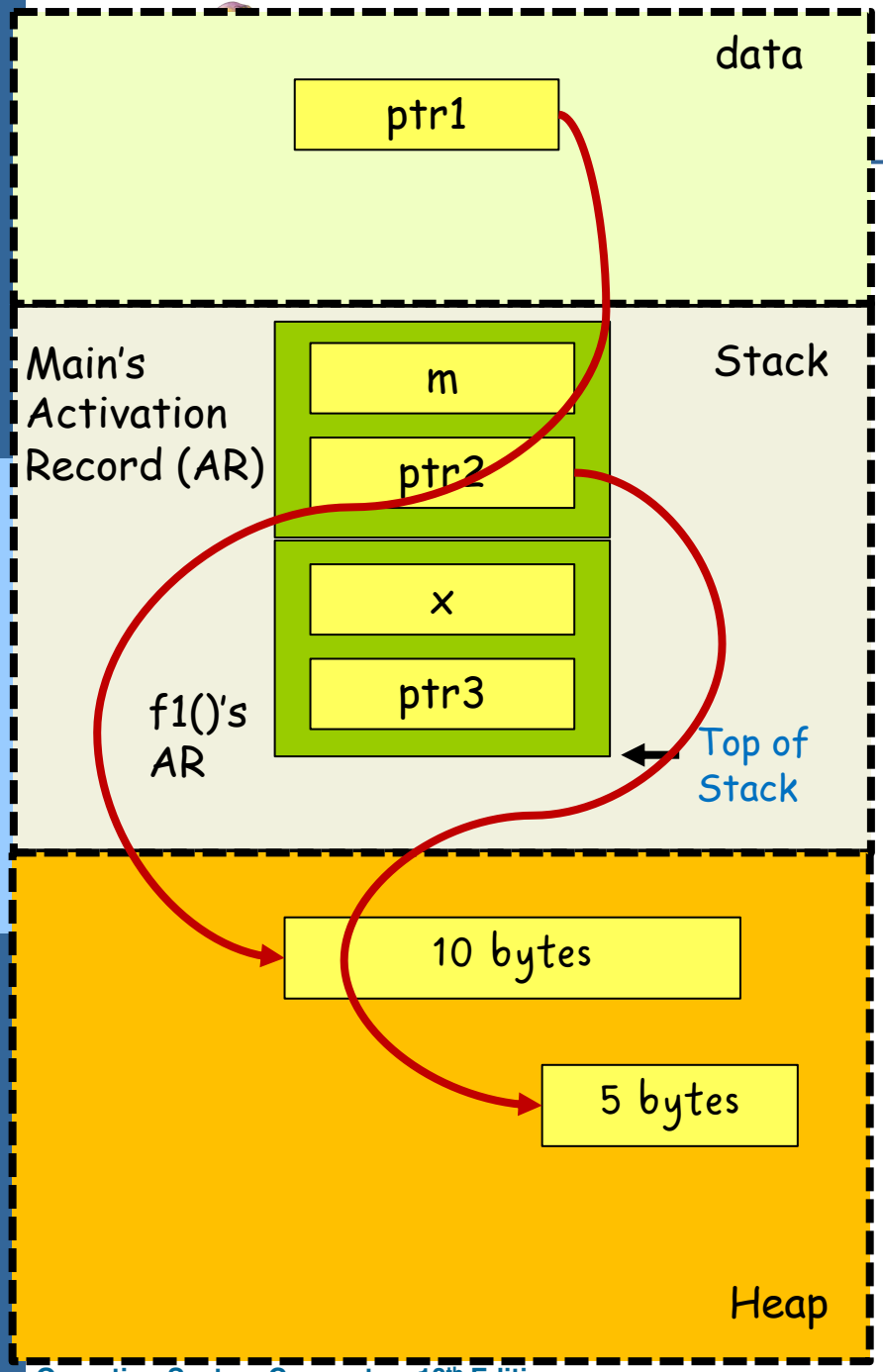



```
char * ptr1
main(){
        int m
        char * ptr2

        ptr1 = malloc(10)
        ptr2 = malloc(5)
        f1();

}

f1(){
        int x
        char * ptr3

        ptr3 = malloc(5)
        printf("hello world")
        free(ptr3)
}
```

Heap

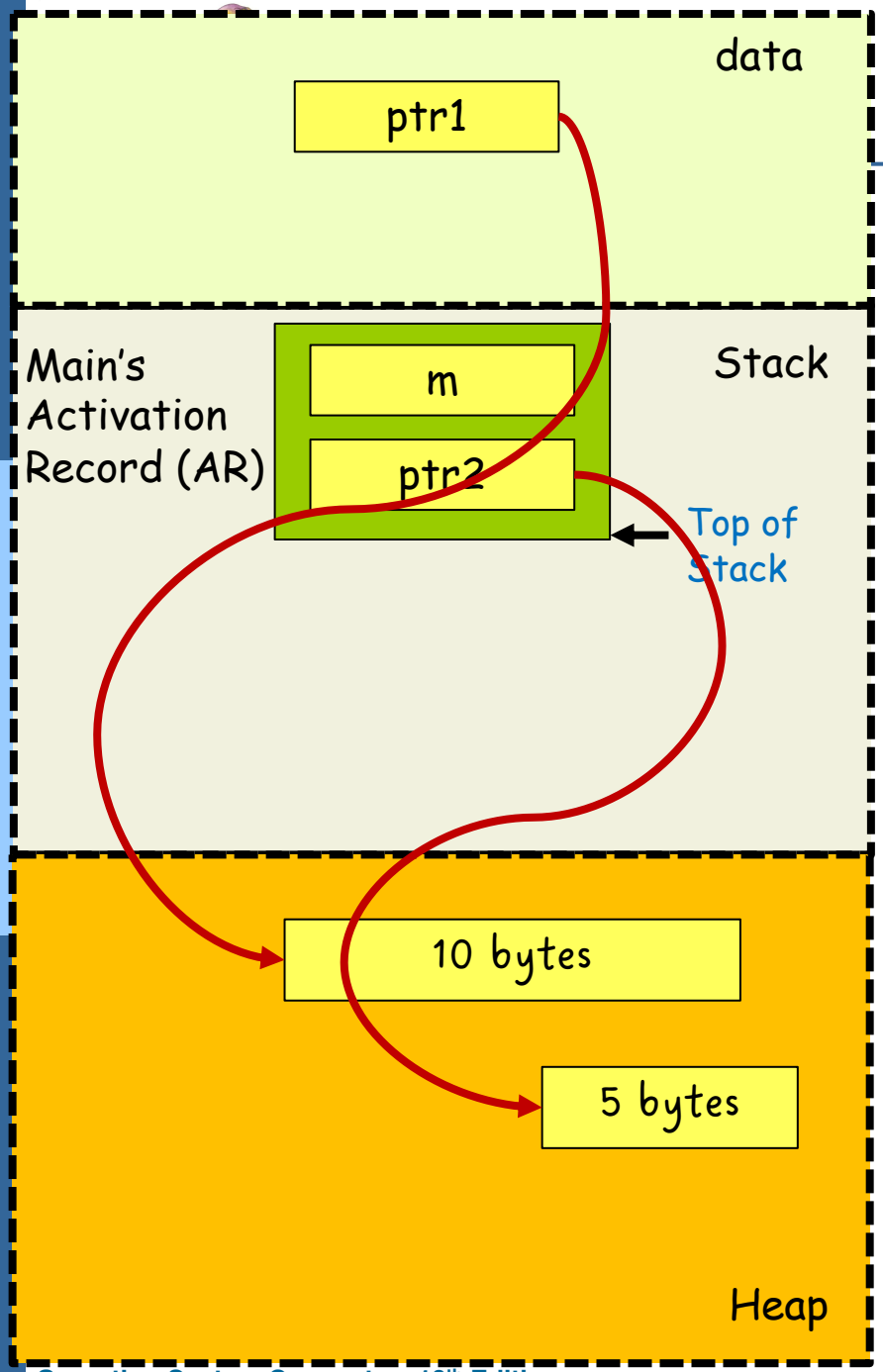



```
char * ptr1
main(){
      int m
      char * ptr2

      ptr1 = malloc(10)
      ptr2 = malloc(5)
      f1();

}

f1(){
      int x
      char * ptr3

      ptr3 = malloc(5)
      printf("hello world")
      free(ptr3)
}
```



Heap

```
char * ptr1
main(){
      int m
      char * ptr2

      ptr1 = malloc(10)
      ptr2 = malloc(5)
      f1();

}

f1(){
      int x
      char * ptr3

      ptr3 = malloc(5)
      printf("hello world")
      free(ptr3)
}
```



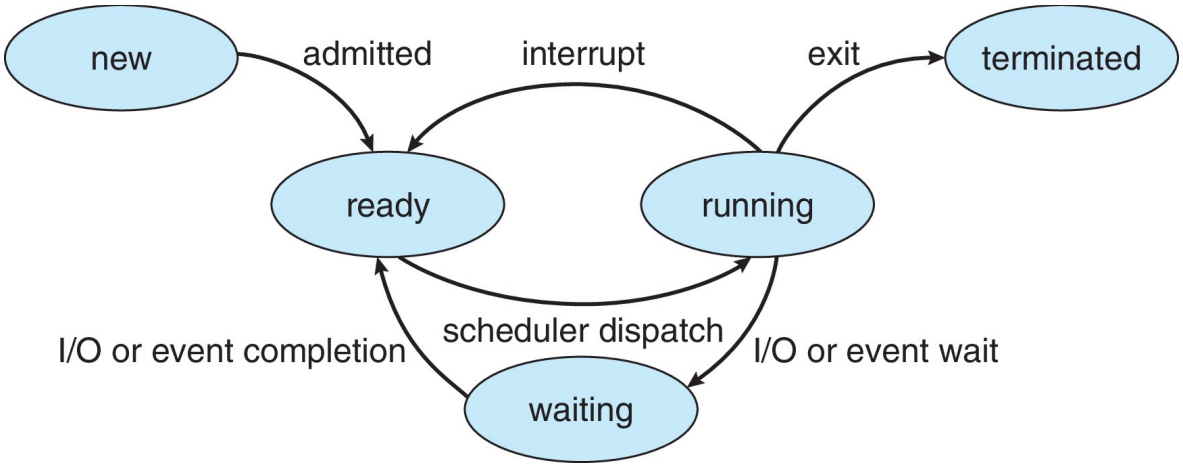
Heap

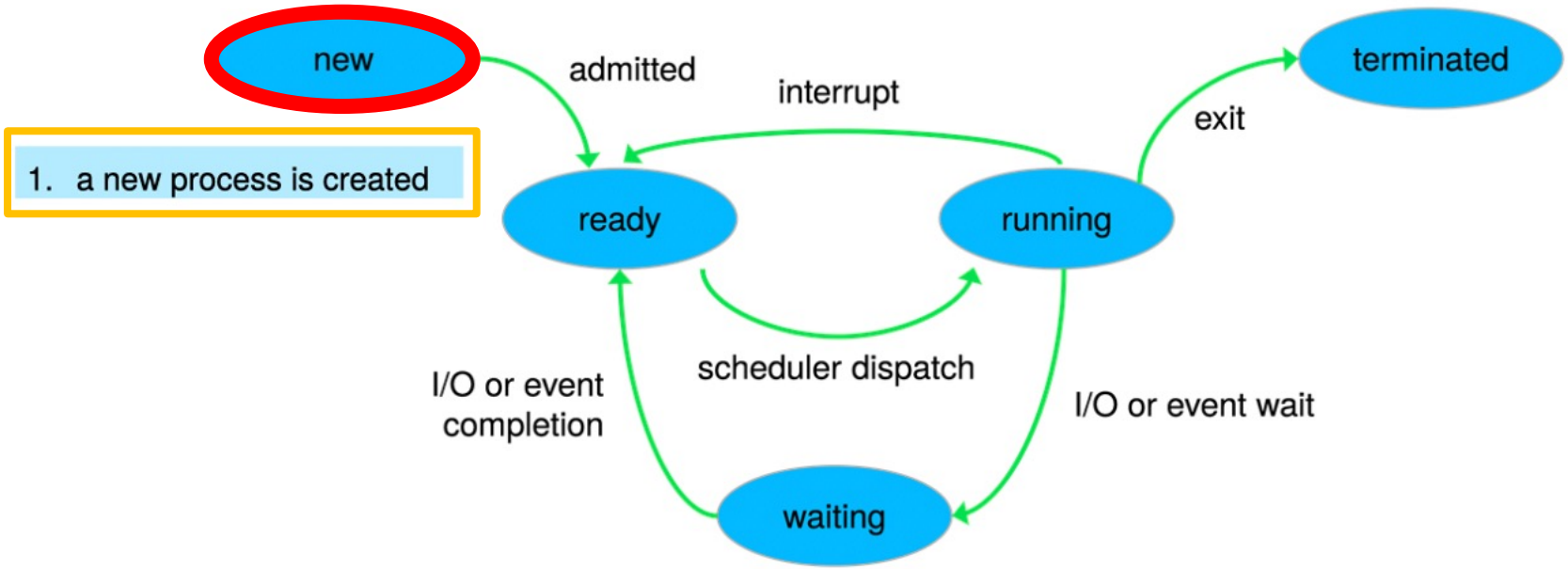
```
char * ptr1
main(){
      int m
      char * ptr2

      ptr1 = malloc(10)
      ptr2 = malloc(5)
      f1();

}

f1(){
      int x
      char * ptr3

      ptr3 = malloc(5)
      printf("hello world")
      free(ptr3)
}
```

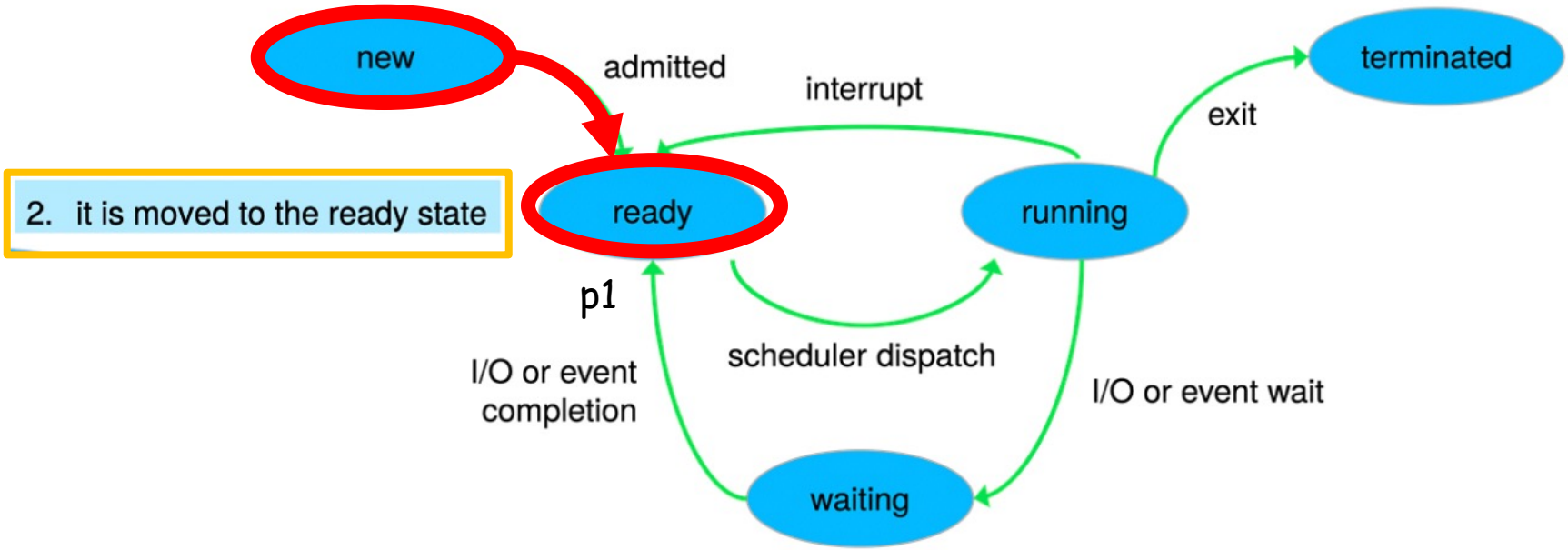
Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

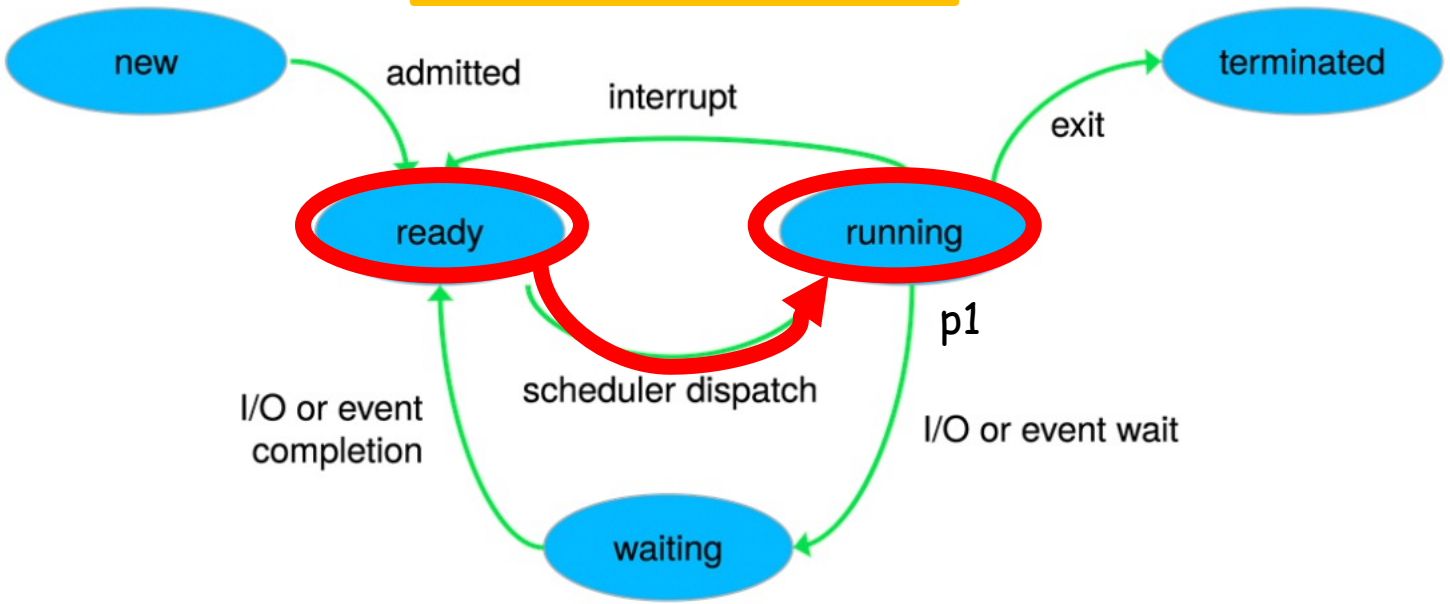
Diagram of Process State

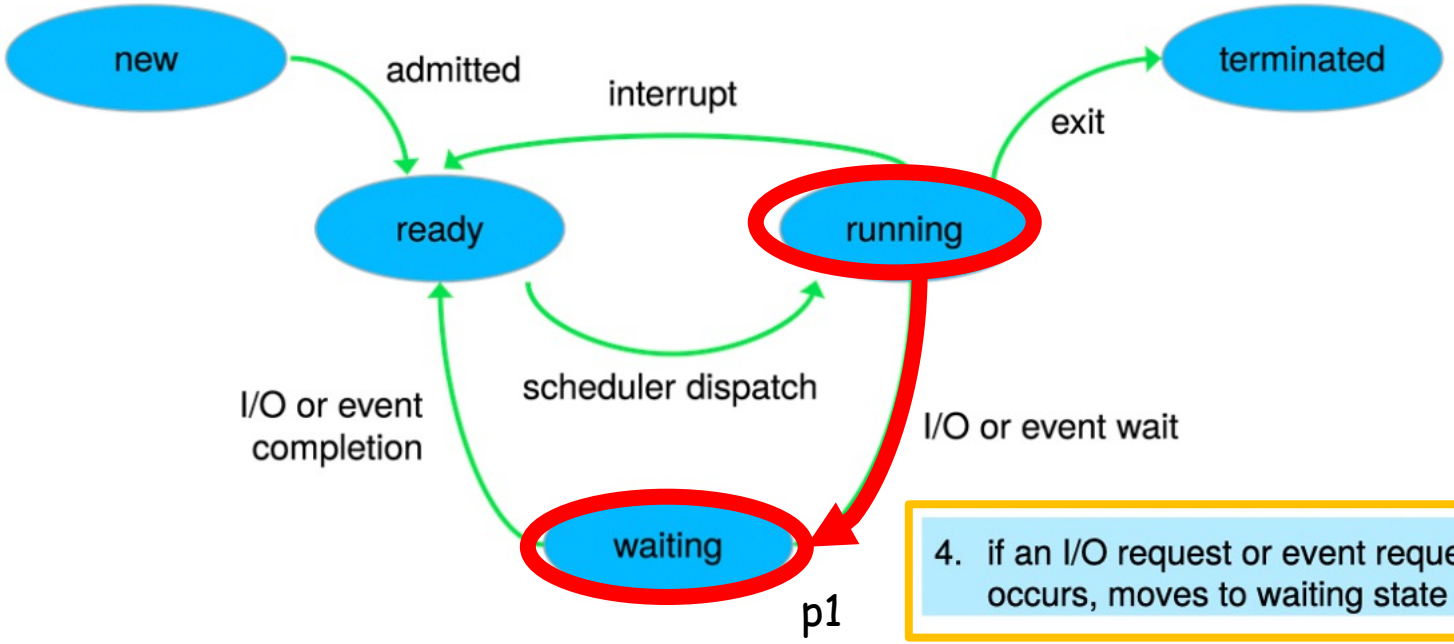


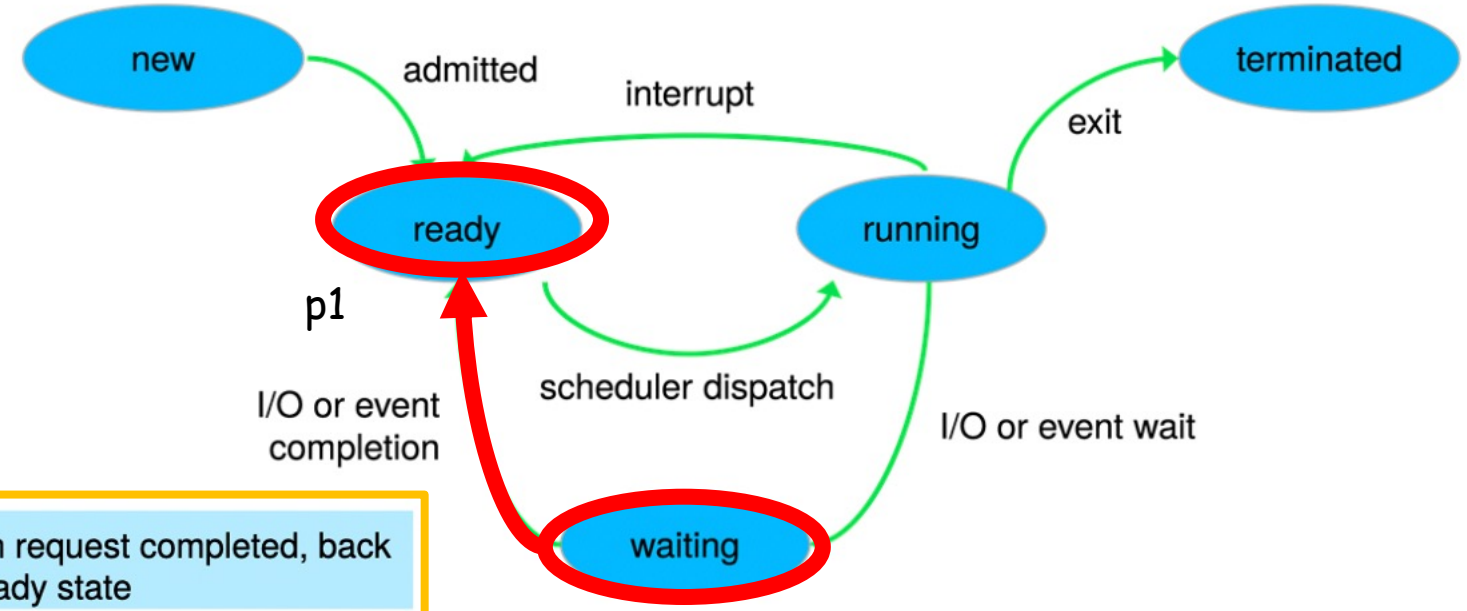

new
admitted
interrupt
terminated
exit
1. a new process is created
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting


new
admitted
interrupt
terminated
exit
2. it is moved to the ready state
ready
running
p1
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

3. it is scheduled onto a core when the core is available and is now running

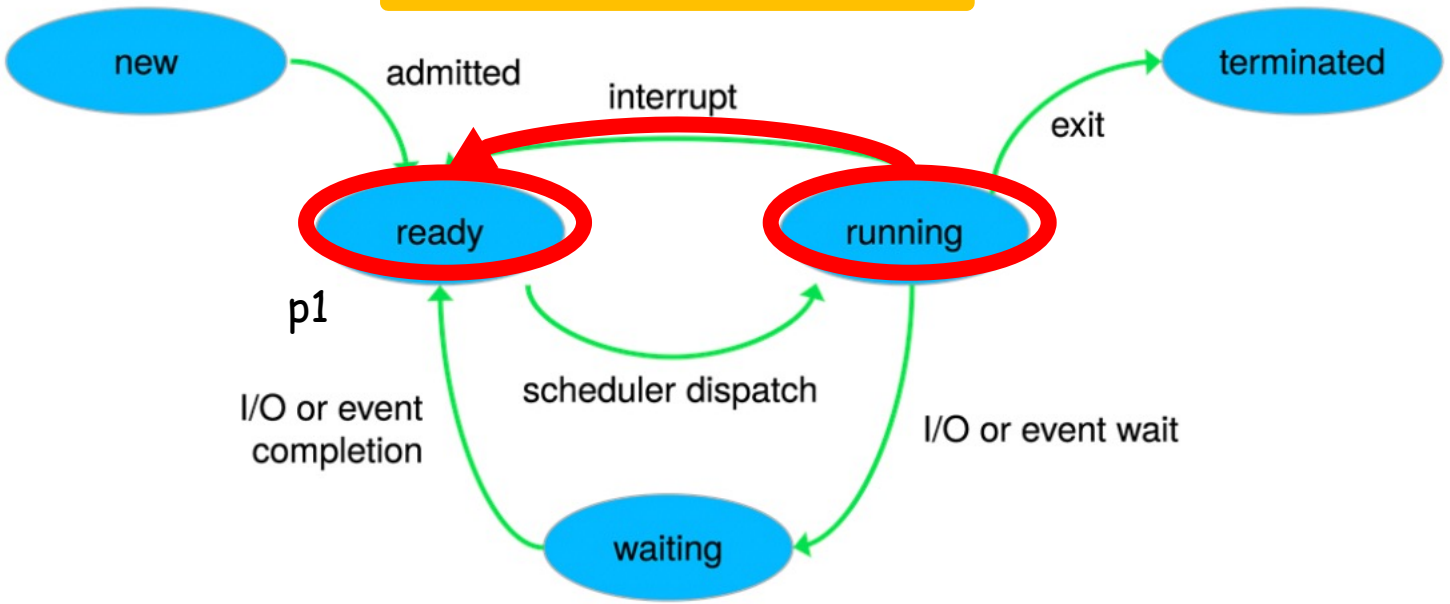



new
admitted
interrupt
terminated
exit
ready
running
scheduler dispatch
I/O or event
completion
I/O or event wait
waiting
p1
4. if an I/O request or event request occurs, moves to waiting state

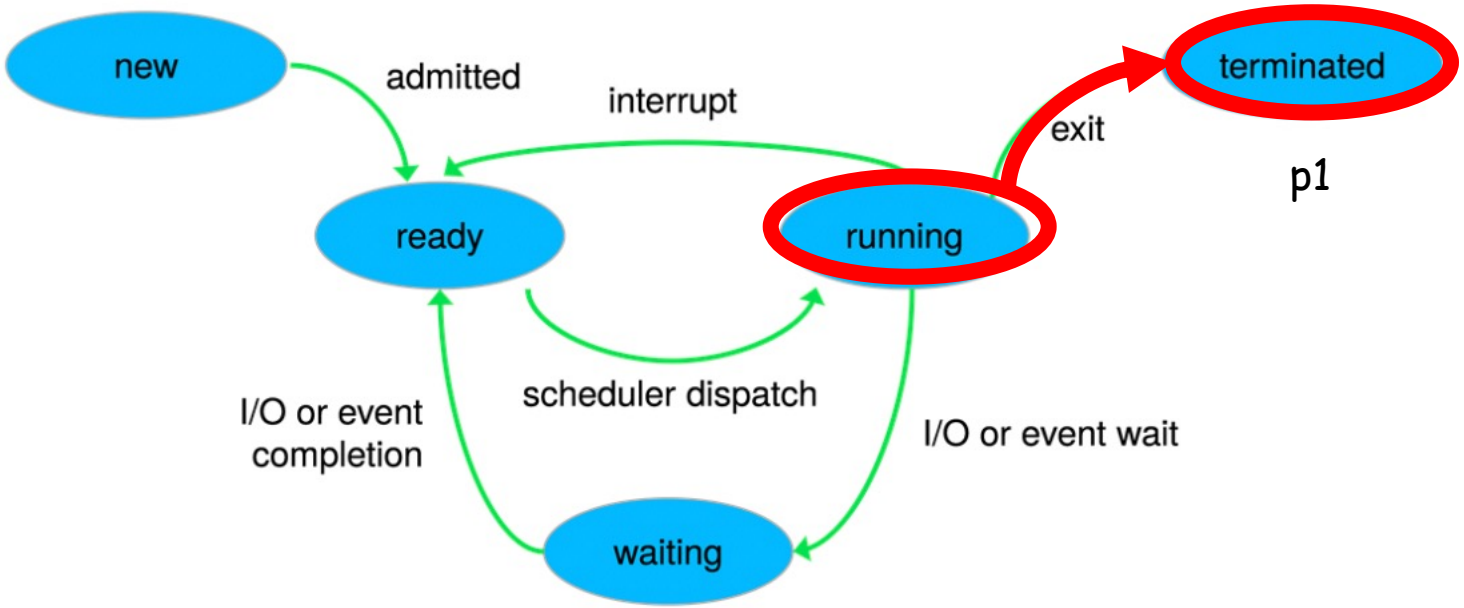


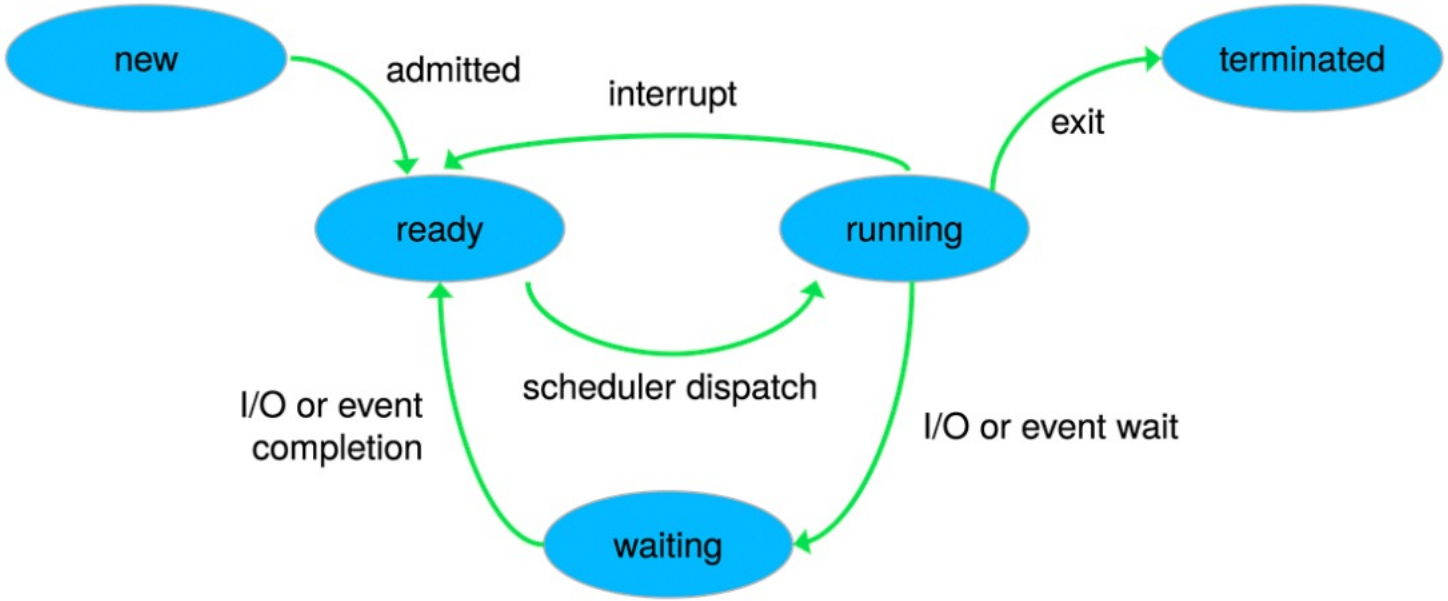
5. when request completed, back to ready state

6. if running and the core is needed (say for an interrupt), back to the ready state



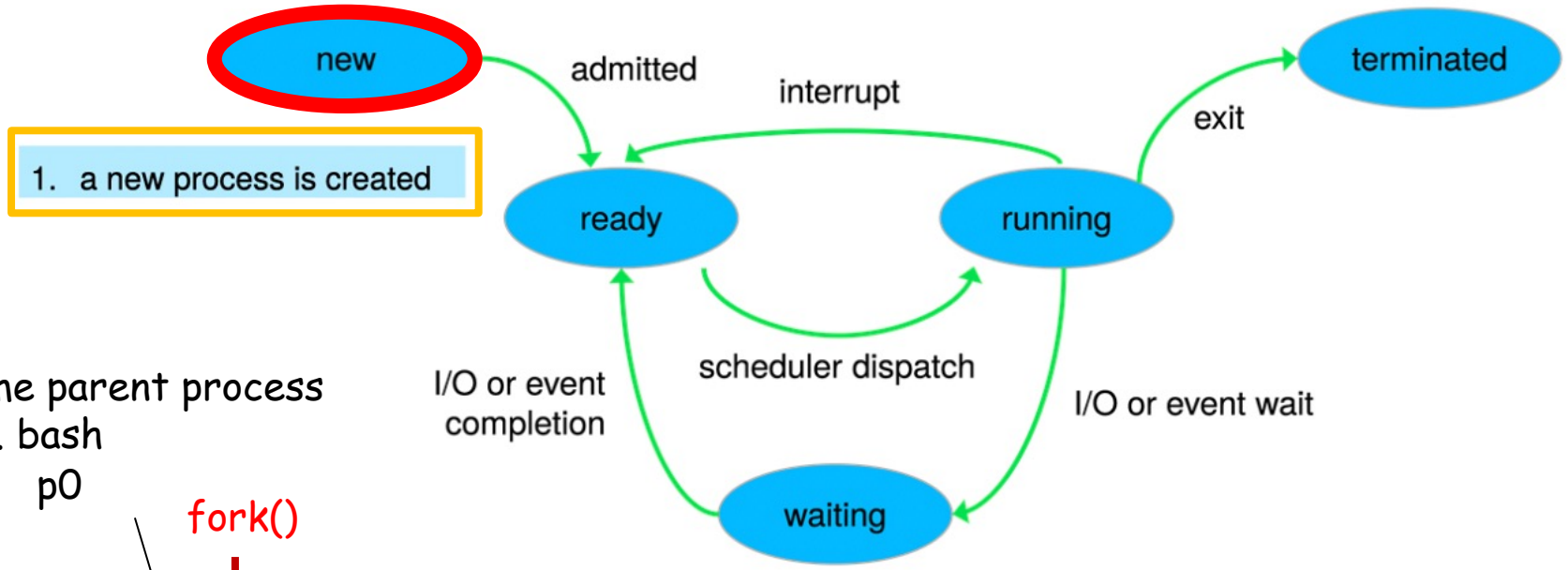
7. cycle continues until the process exists, fails, or is terminated, moves to terminated state



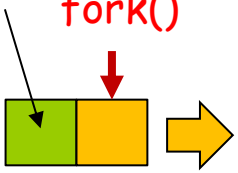

new
admitted
interrupt
terminated
exit
ready
running
scheduler dispatch
I/O or event
completion
I/O or event wait
waiting

OS

p0



some parent process
e.g. bash
p0

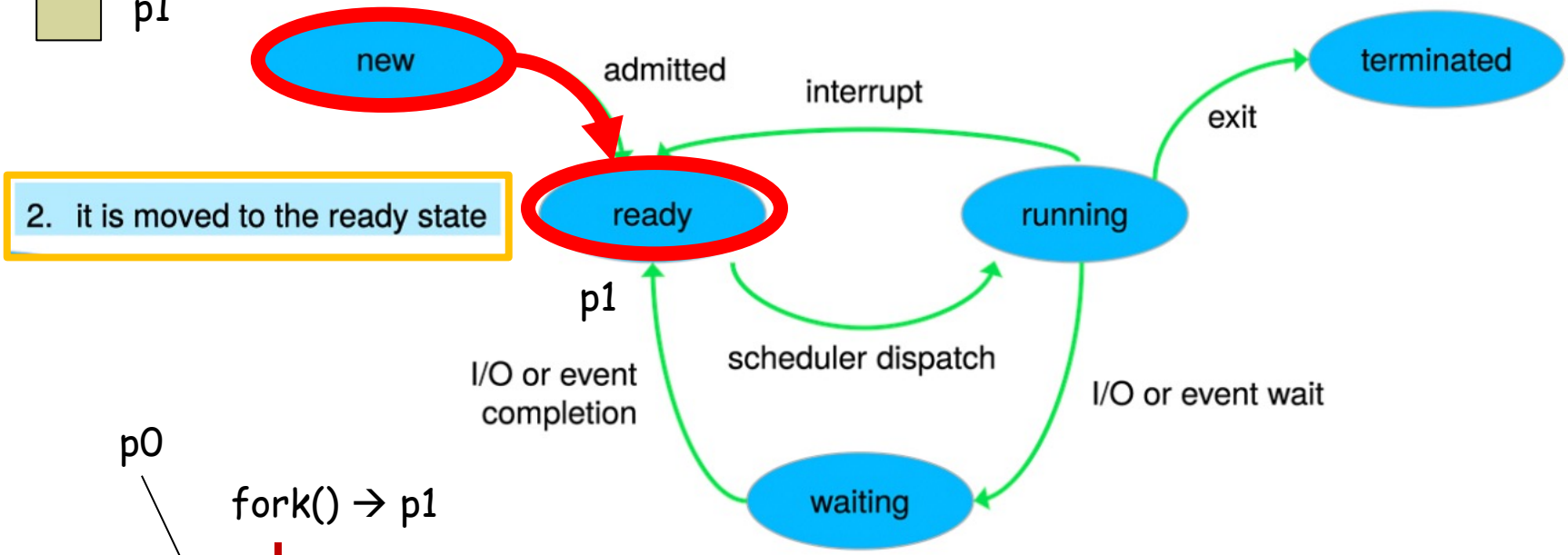


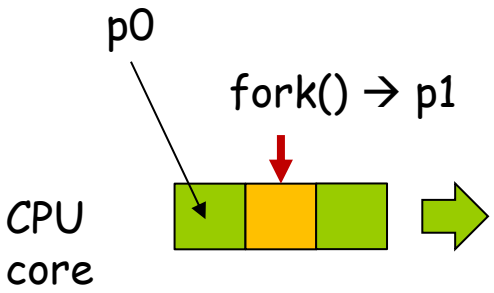
CPU core

OS

p0

p1


new
admitted
interrupt
terminated
exit
2. it is moved to the ready state
ready
running
p1
I/O or event completion
scheduler dispatch
I/O or event wait
waiting


p0
fork() → p1
CPU core

OS

p0

p1

3. it is scheduled onto a core when the core is available and is now running

new

admitted

interrupt

exit

terminated

ready

running

p1

scheduler dispatch

I/O or event completion

I/O or event wait

waiting

some events happened
And OS get control of CPU core

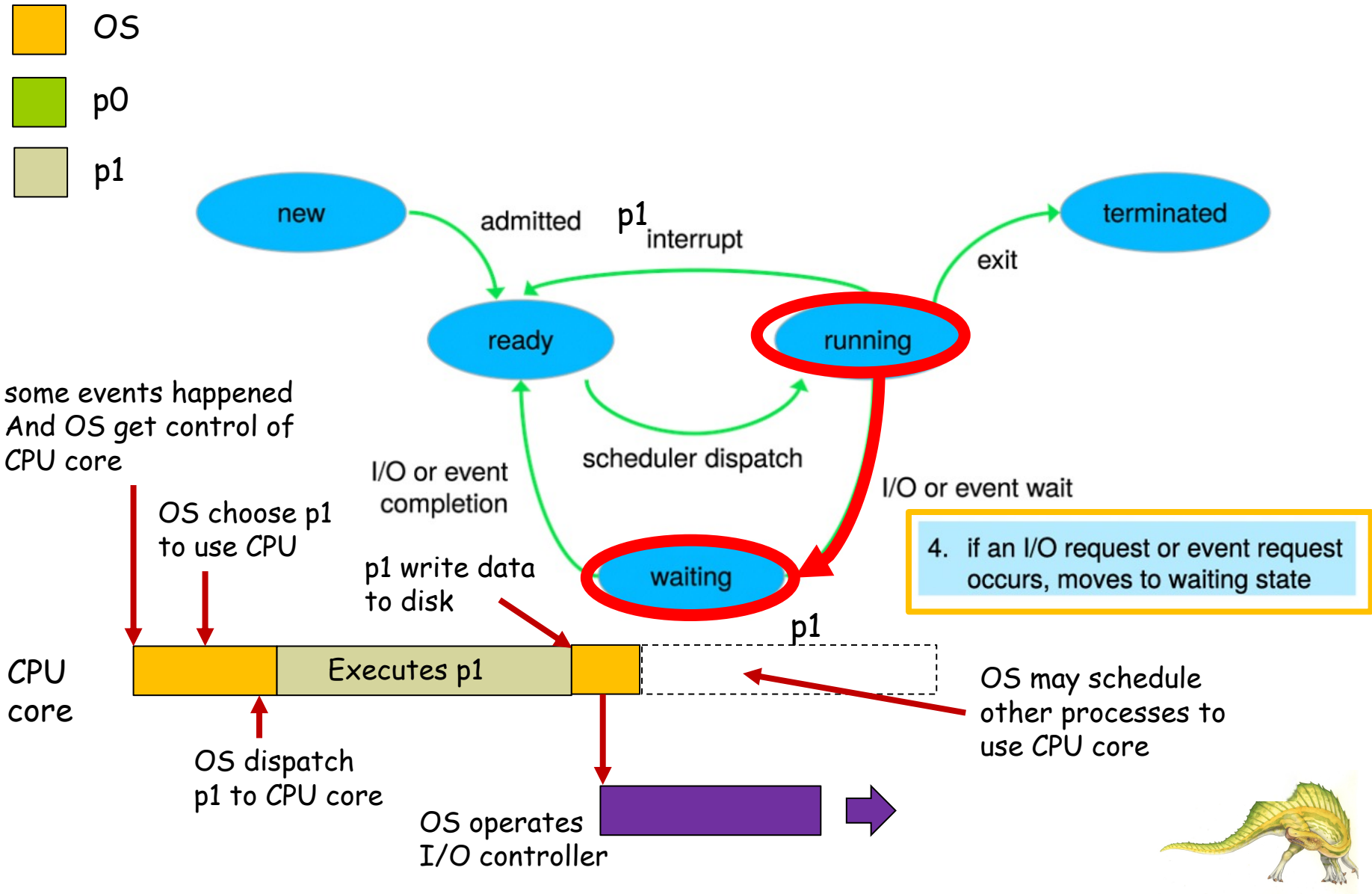
OS choose p1 to use CPU

CPU core

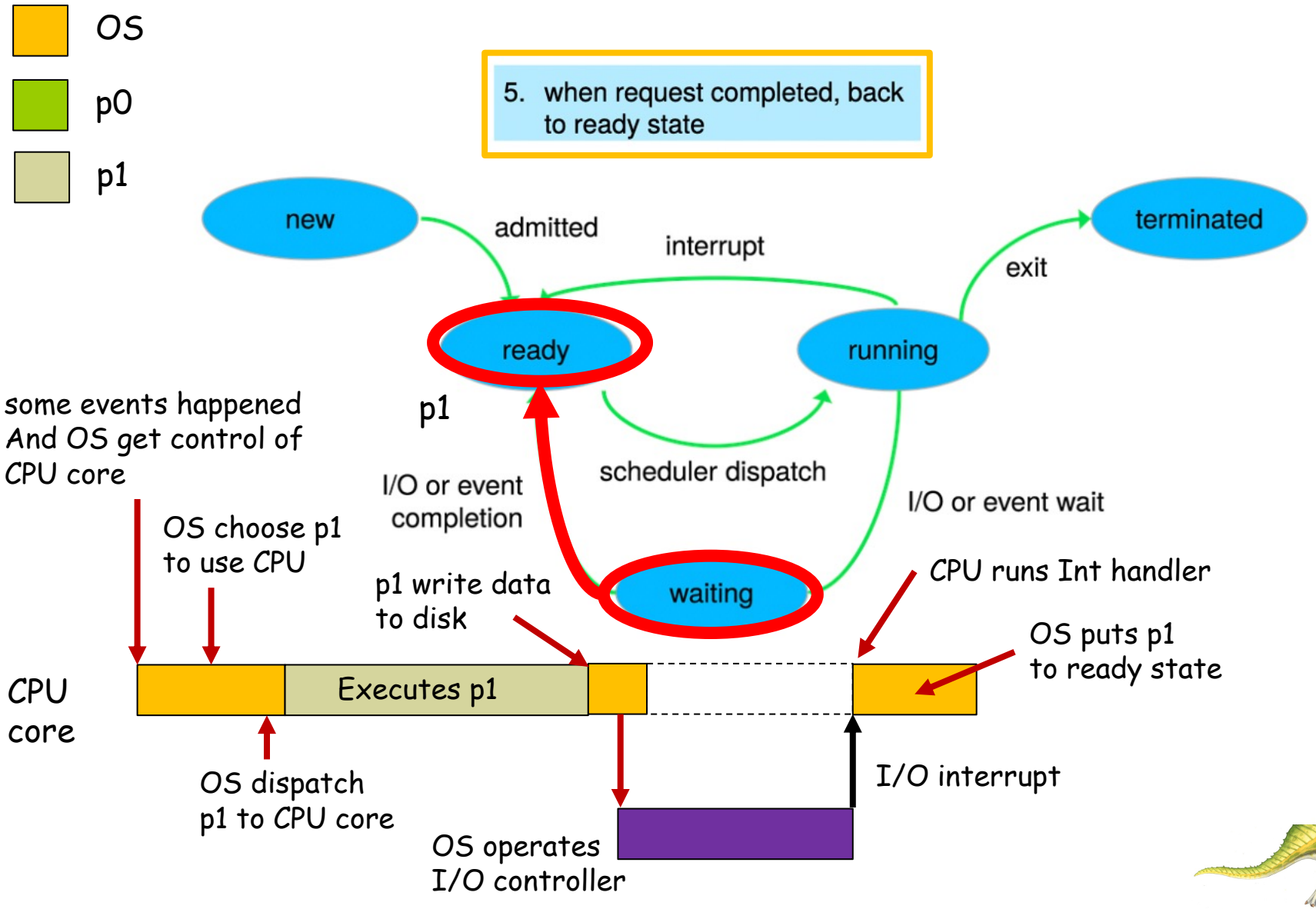
OS dispatch p1 to CPU core

CPU core executes p1

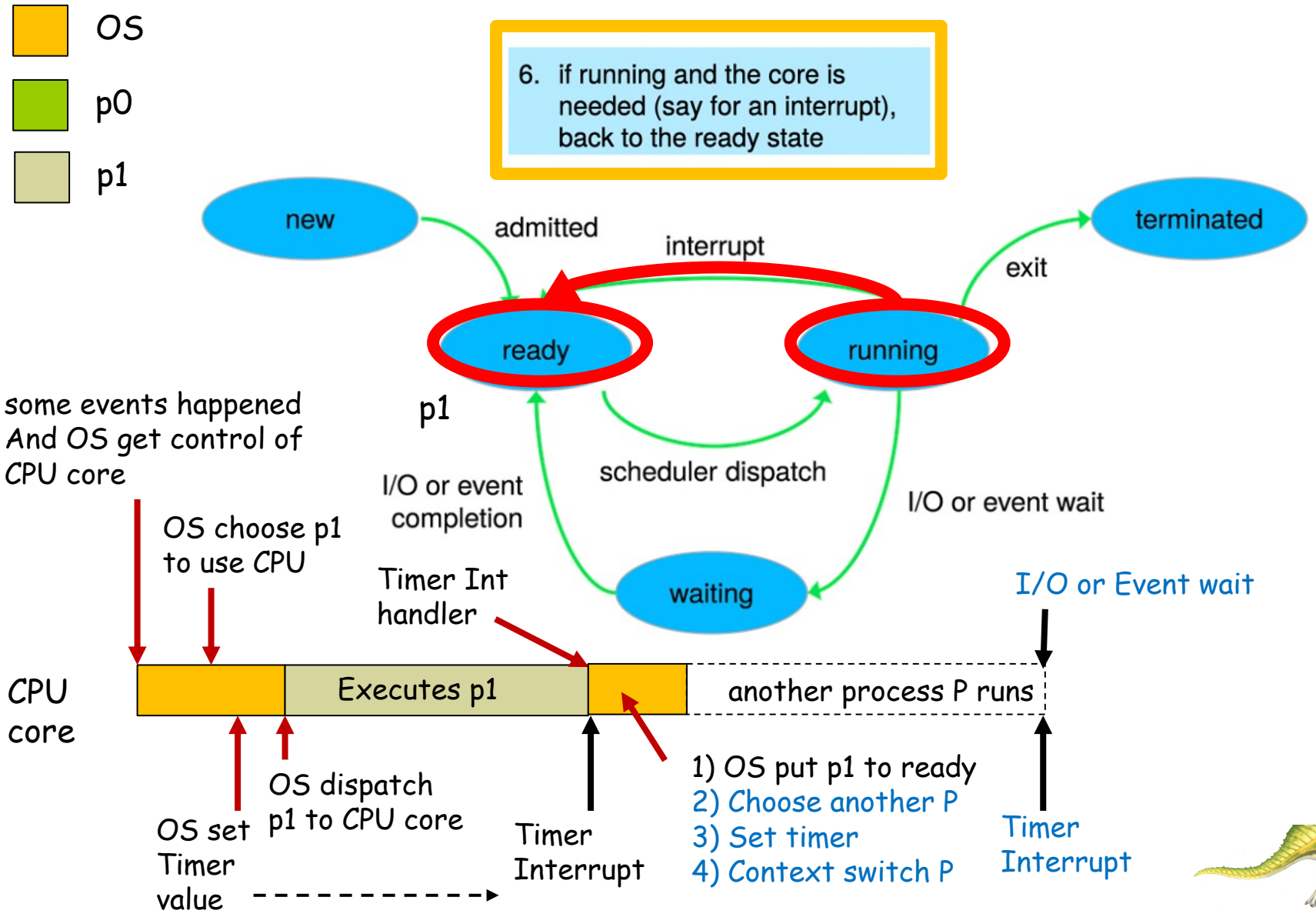
กรณีที่ 1: p1 รันและใช้ I/O (1)



กรณีที่ 1: ถ้า p1 รันและใช้ I/O (2)



กรณีที่ 2: p1 รันและ Time Slice หมด



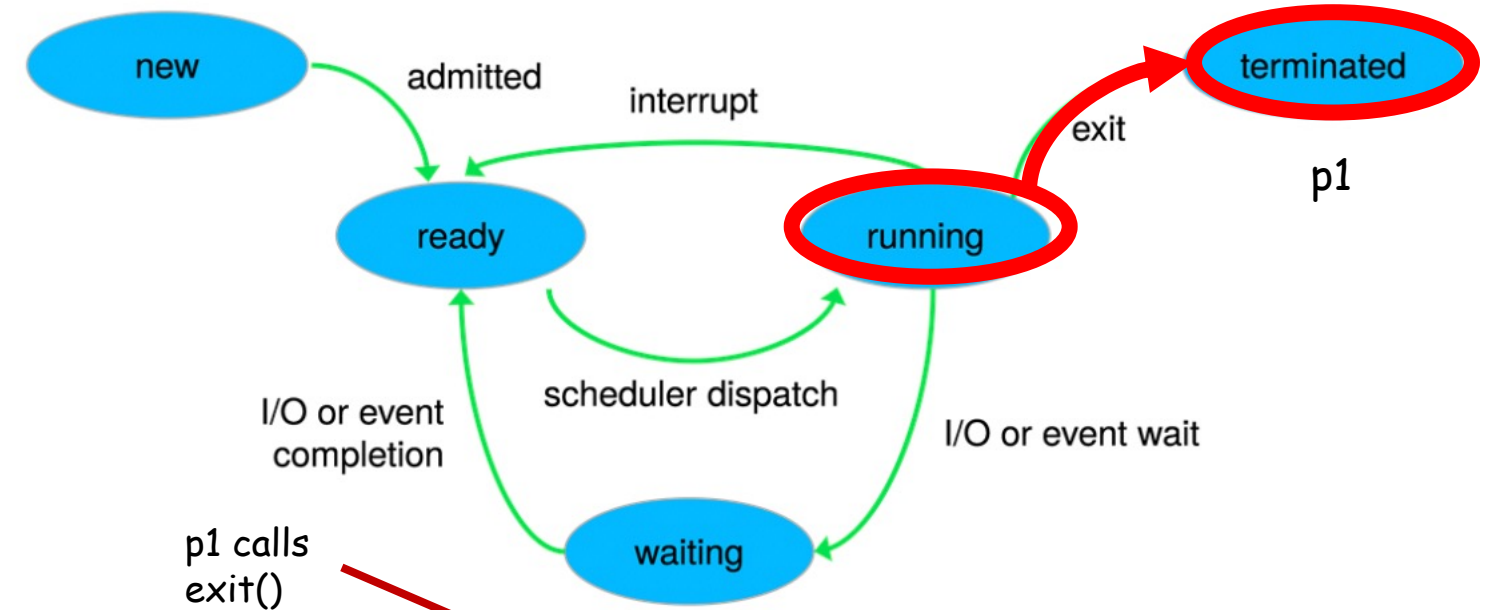
กรณีที่ 3: p1 รันและจบการทำงาน exit

OS

p0

p1

7. cycle continues until the process exists, fails, or is terminated, moves to terminated state



p1 calls exit()

CPU core

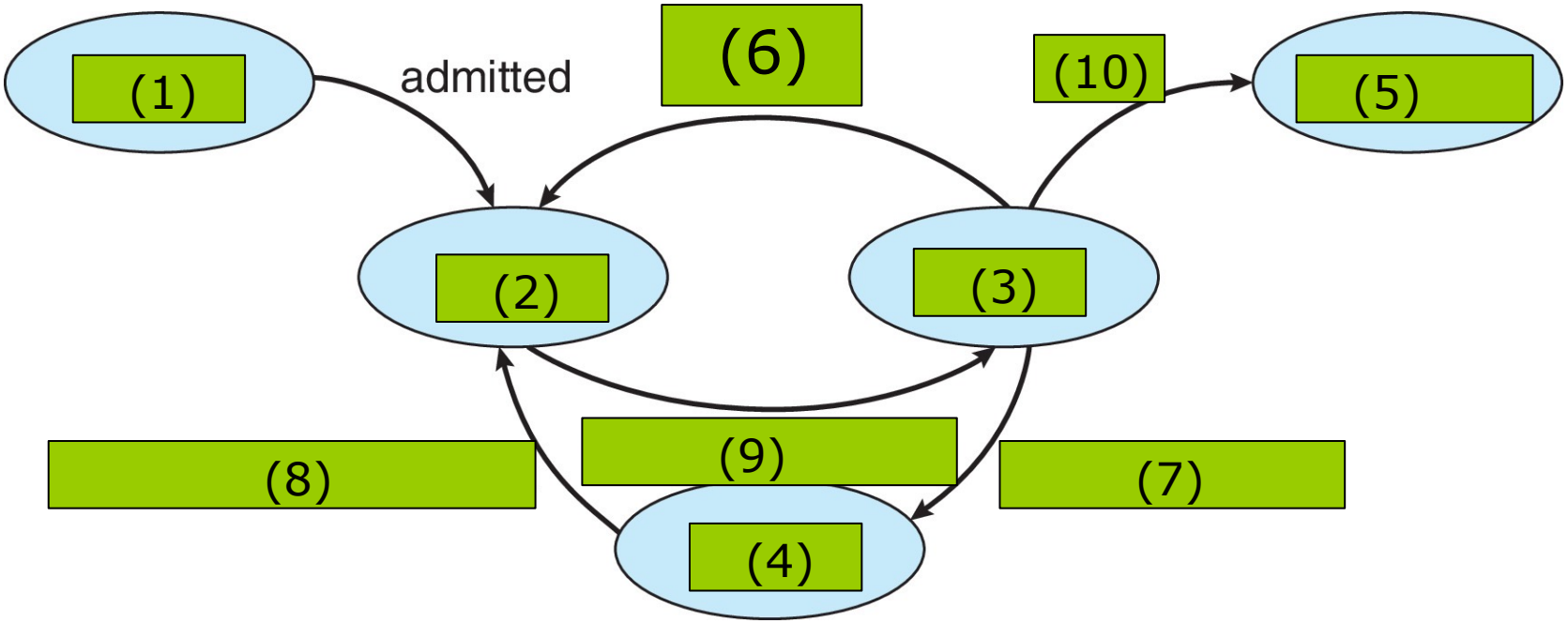
Executes p1

another process P runs

OS put p1 to terminated

1) OS put p1 to ready
2) Choose another P
3) Set timer
4) Context switch P

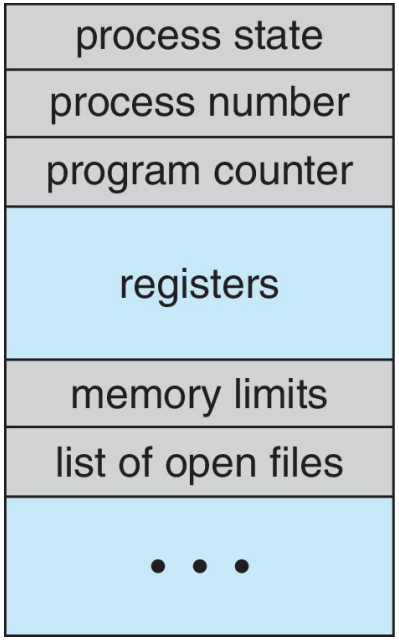
Diagram of Process State (pre-test)



Process Control Block (PCB)

Information associated with each process(also called **task control block**)

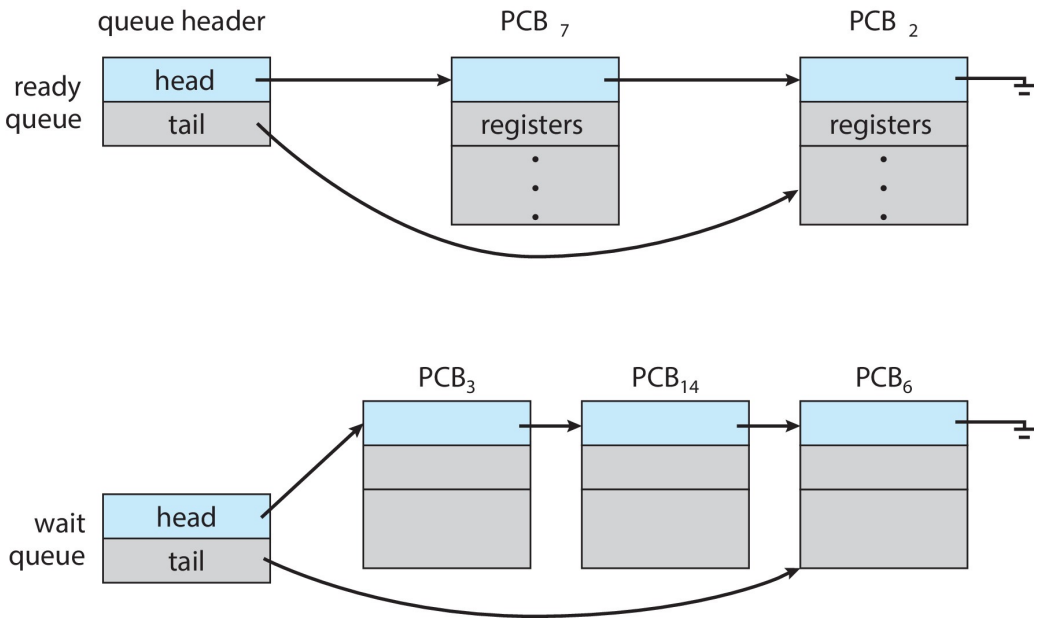
- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



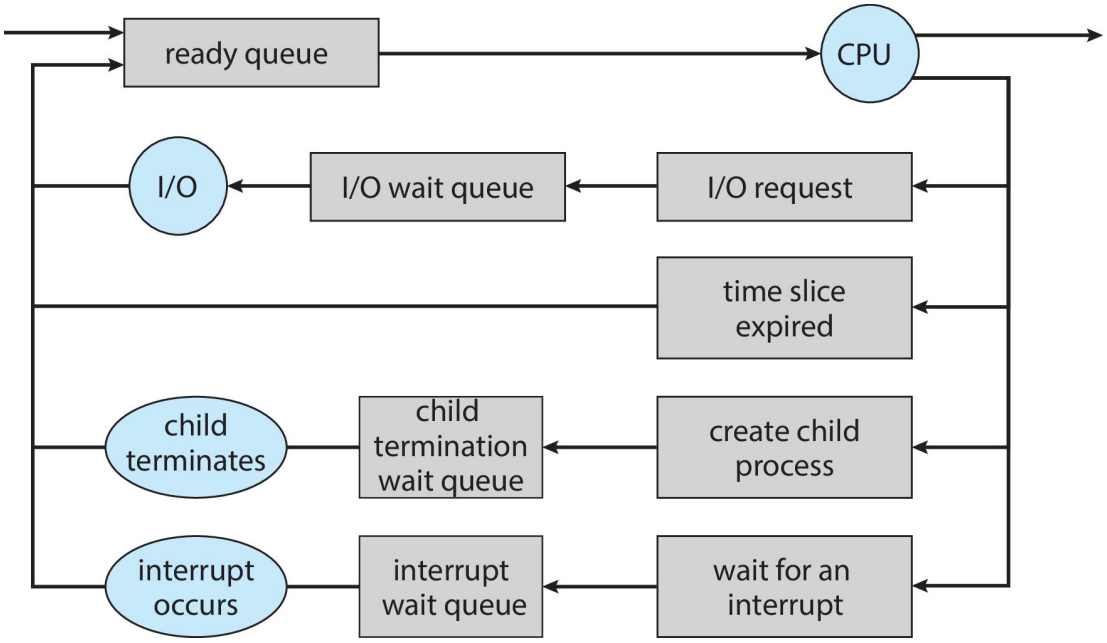
Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e., I/O)
  - Processes migrate among the various queues
- **Degree of Multiprogramming** is the number of processes currently in memory

Ready and Wait Queues



Representation of Process Scheduling



สถานการณ์จำลอง

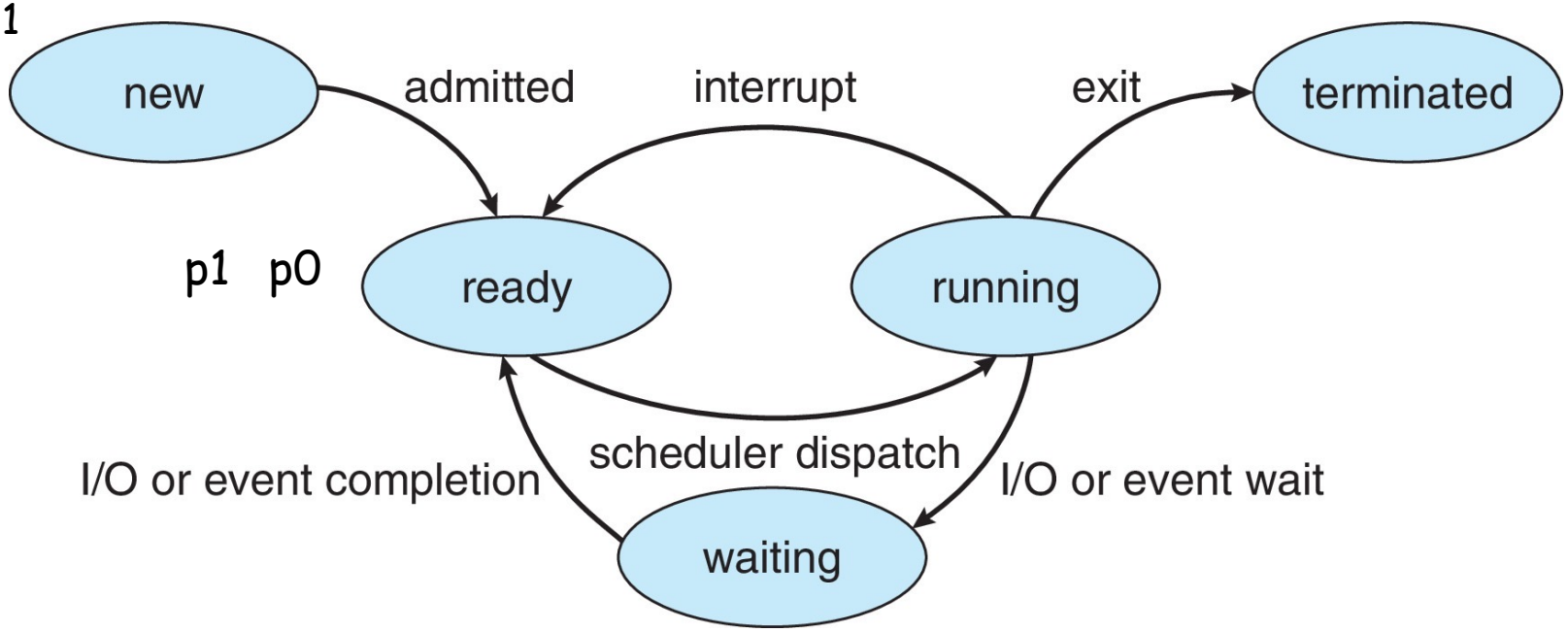
- สมมุติว่ามี Process อยู่ 2 Process ได้แก่ p0 และ p1 และ
- degree of multiprogramming = 2
- OS's Process Scheduler จะเลือก Process เข้ามาใช้งาน CPU

Diagram of Process State

OS

p0

p1



CPU

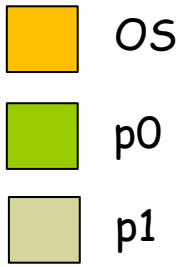
Representation of Process Scheduling

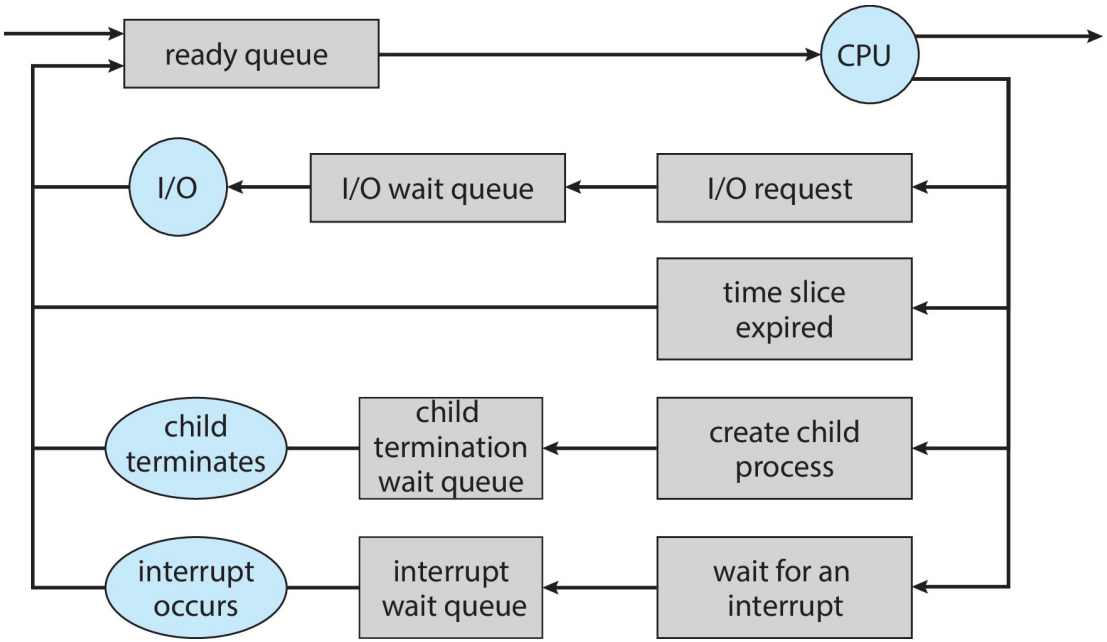
โปรเซสที่มีอยู่ในระบบ

p0, p1

OS's Process scheduler

p1 p0





CPU

สถานการณ์จำลอง

- Process Scheduler (โค้ดของ OS kernel) จะเลือก Process P0 เข้ามาใช้งาน CPU

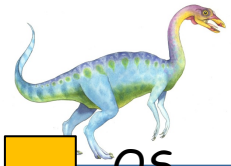
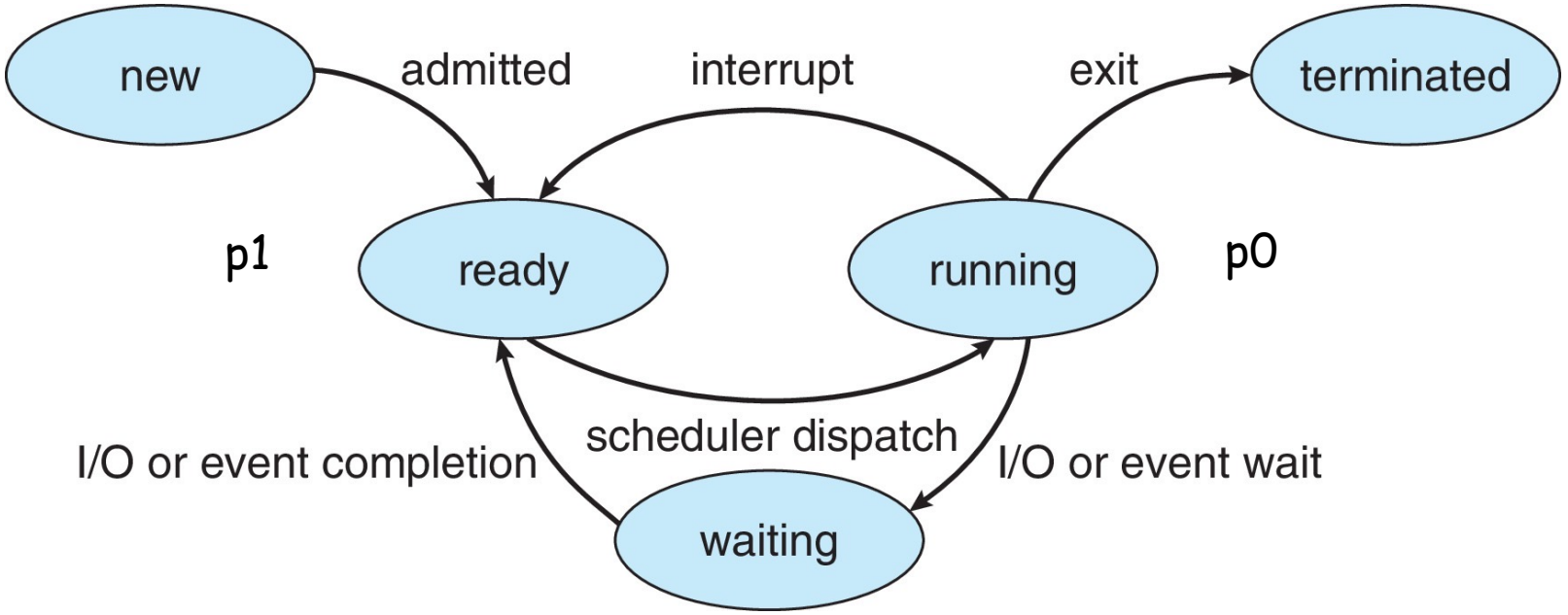


Diagram of Process State

OS

p0

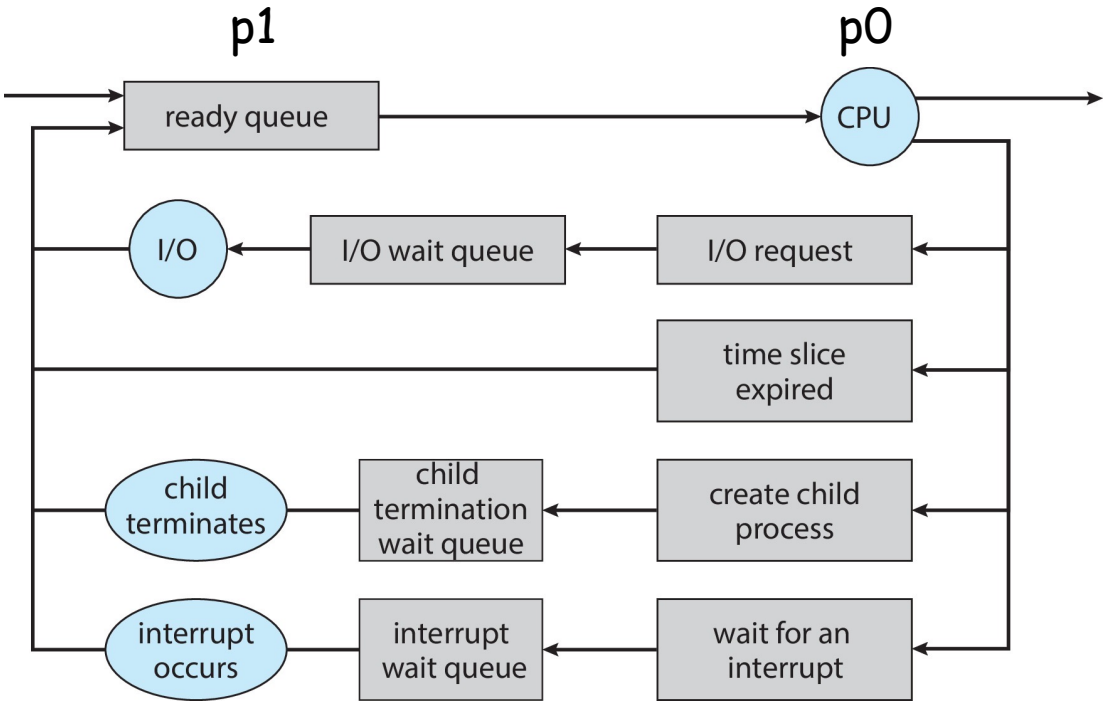
p1



CPU

Representation of Process Scheduling

Event: Dispatch p0

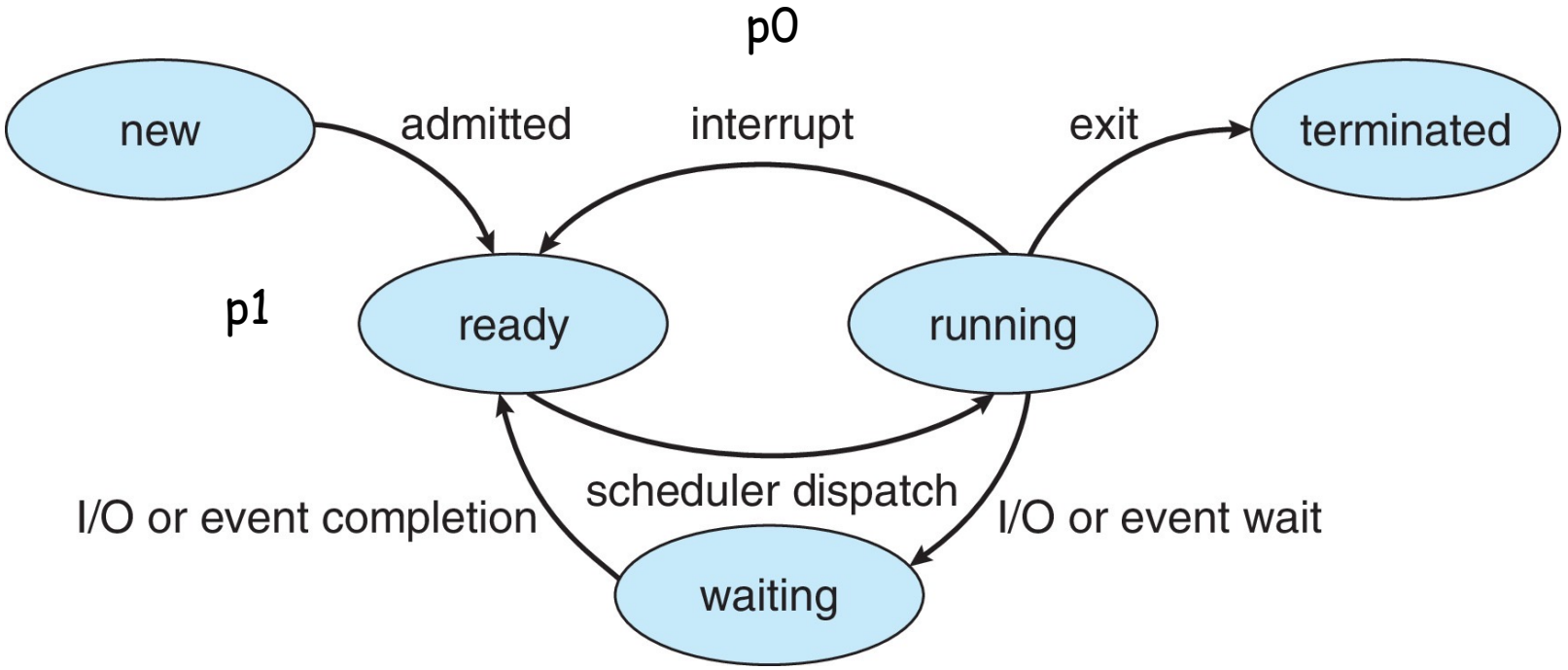


CPU 

สถานการณ์จำลอง

- Time Slice ของ P0 หมดเวลา
- Hardware Timer ส่งสัญญาณ Timer Interrupt ไปยัง CPU
- CPU รัน Timer Interrupt Handler (โค๊ดของ OS)
  - Save context ของ P0 ไป PCB0
  - ย้าย P0 ไป ready queue (เปลี่ยน state ของ P0)

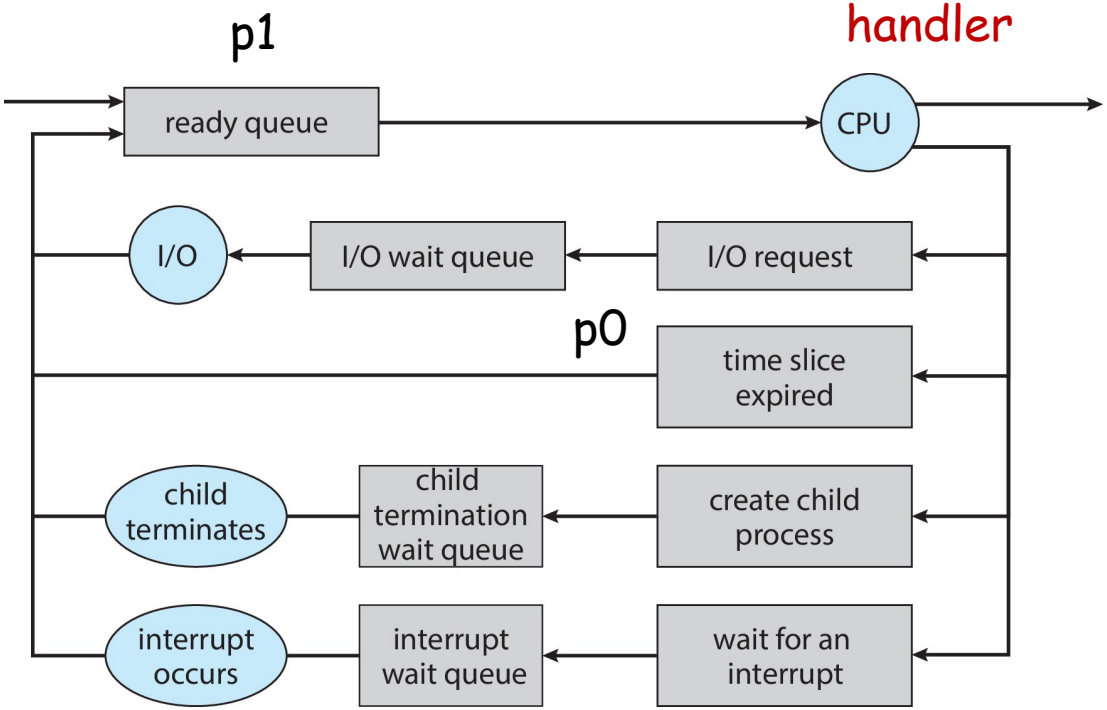
Diagram of Process State



Representation of Process Scheduling

Event: Timer int because p0's time slice expires

OS
Timer Int
handler



Timer Int handler is a part of OS

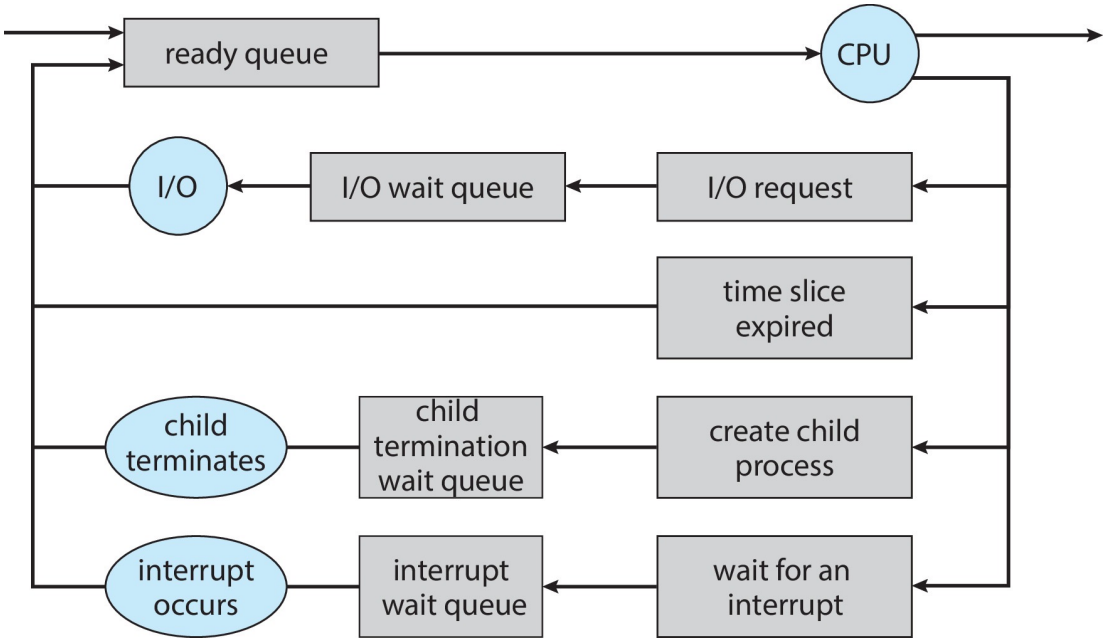
CPU

Representation of Process Scheduling

Event: Timer int because p0's time slice expires

OS Process scheduler

p0 p1

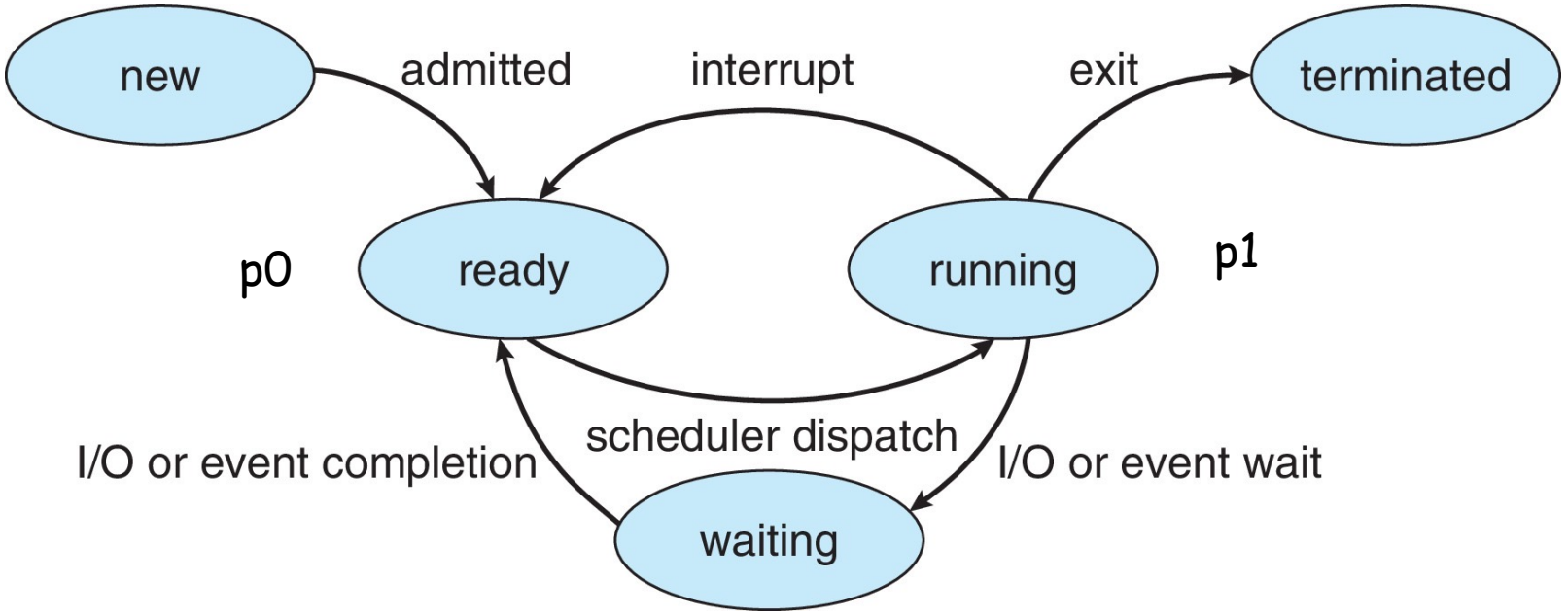


CPU

สถานการณ์จำลอง

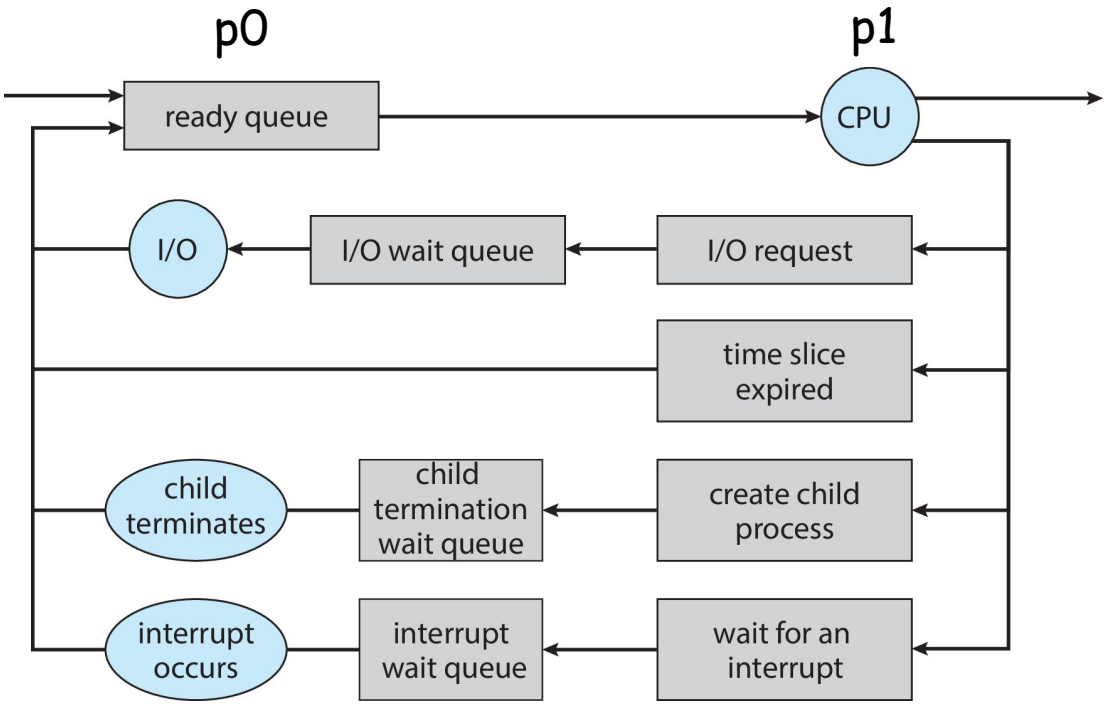
- Time Slice ของ P0 หมดเวลา
- Hardware Timer ส่งสัญญาณ Timer Interrupt ไปยัง CPU
- CPU รัน Timer Interrupt Handler (โค๊ดของ OS)
  - Save context ของ P0 ไป PCB0
  - ย้าย P0 ไป ready queue (เปลี่ยน state ของ P0)
  - รัน Process Scheduler เพื่อเลือก process ใหม่ (p1)
  - Restore context ของ P1 จาก PCB1 มายัง CPU registers
  - ลบ p1 จาก ready queue เปลี่ยน state
  - ปล่อยให้ p1 ประมวลผลต่อ

Diagram of Process State



Representation of Process Scheduling

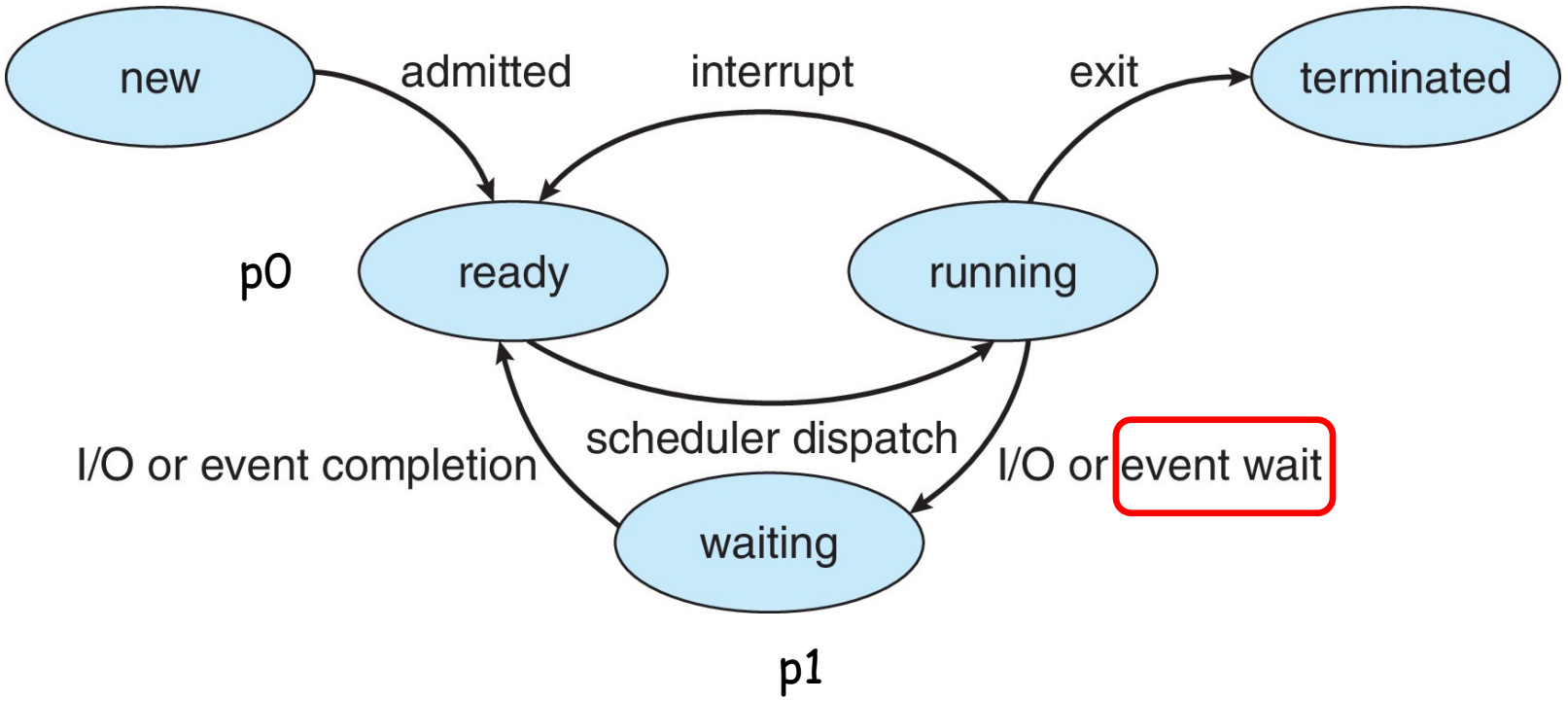
Event: OS dispatch p1 to use CPU core



สถานการณ์จำลอง

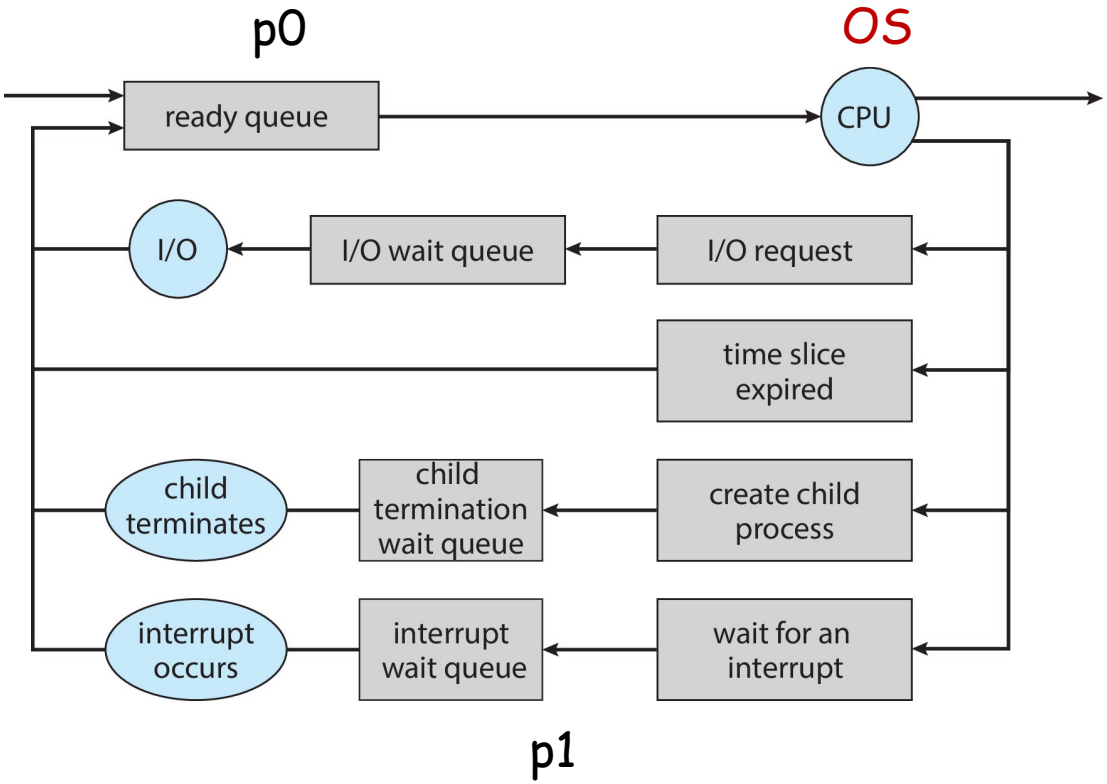
- P1 เรียก read system call เพื่ออ่านข้อมูลจาก Disk *ซึ่งต้องใช้เวลานาน เนื่องจากข้อมูลมีขนาดใหญ่*
- read() system call (in OS kernel) ประมวลผล
  - Save context ของ P1 ไป PCB1
  - นำ P1 ไปรอใน Interrupt Wait Queue (เปลี่ยน state ของ P1) (ทุกครั้งที่มี I/O interrupt เกิดขึ้น I/O Interrupt handler จะเข้ามาดูใน Interrupt Wait Queue เพื่อหาข้อมูลของ Process ที่รอ Interrupt นั้นอยู่)

Diagram of Process State



Representation of Process Scheduling

Event: p1 use system call to read data from disk and wait for INTerrupt



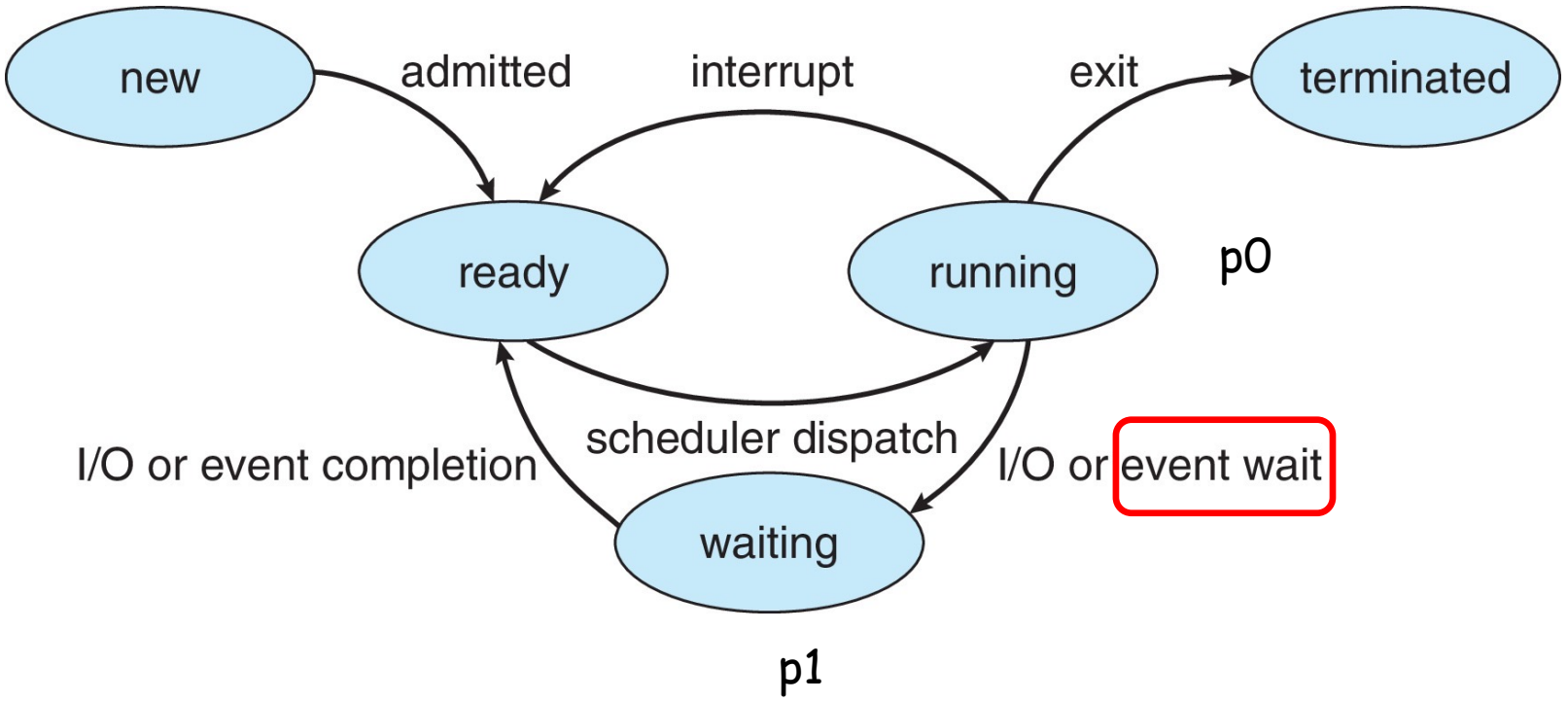
System call is a part of OS

สถานการณ์จำลอง

- P1 เรียก read system call เพื่ออ่านข้อมูลจาก Disk *ซึ่งต้องใช้เวลานาน เนื่องจากข้อมูลมีขนาดใหญ่*
- read() system call (in OS kernel) ประมวลผล
  - Save context ของ P1 ไป PCB1
  - นำ P1 ไปรอใน Interrupt Wait Queue (เปลี่ยน state ของ P1)
  - รัน **Process Scheduler** เพื่อเลือก process ใหม่ (p0)
  - Restore context ของ P0 จาก PCB0 มายัง CPU registers
  - ลบ p0 จาก ready queue และเปลี่ยน state ของ p0 เป็น running
  - ปล่อยให้ p0 ประมวลผลต่อ

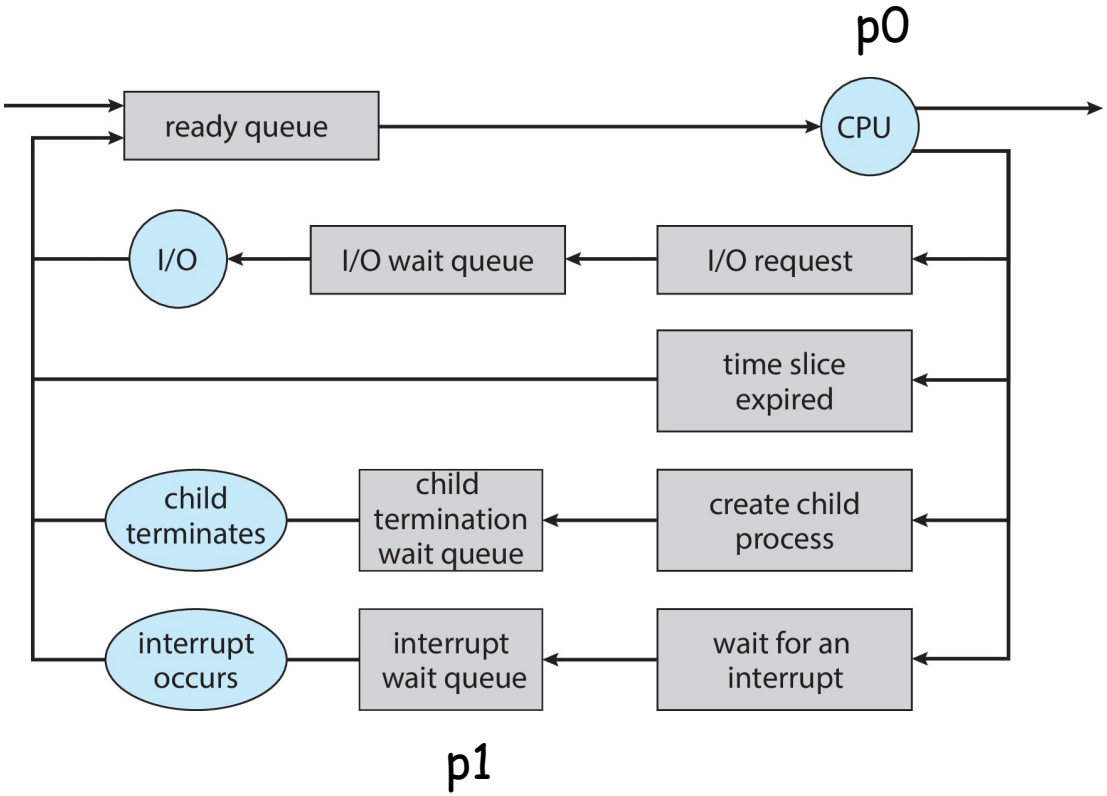
เฉพาะ system call ที่รู้ว่าจะต้องรอ I/O นานและต้องย้าย p1 ไป waitQ เท่านั้นที่จะเรียก schedler

Diagram of Process State



Representation of Process Scheduling

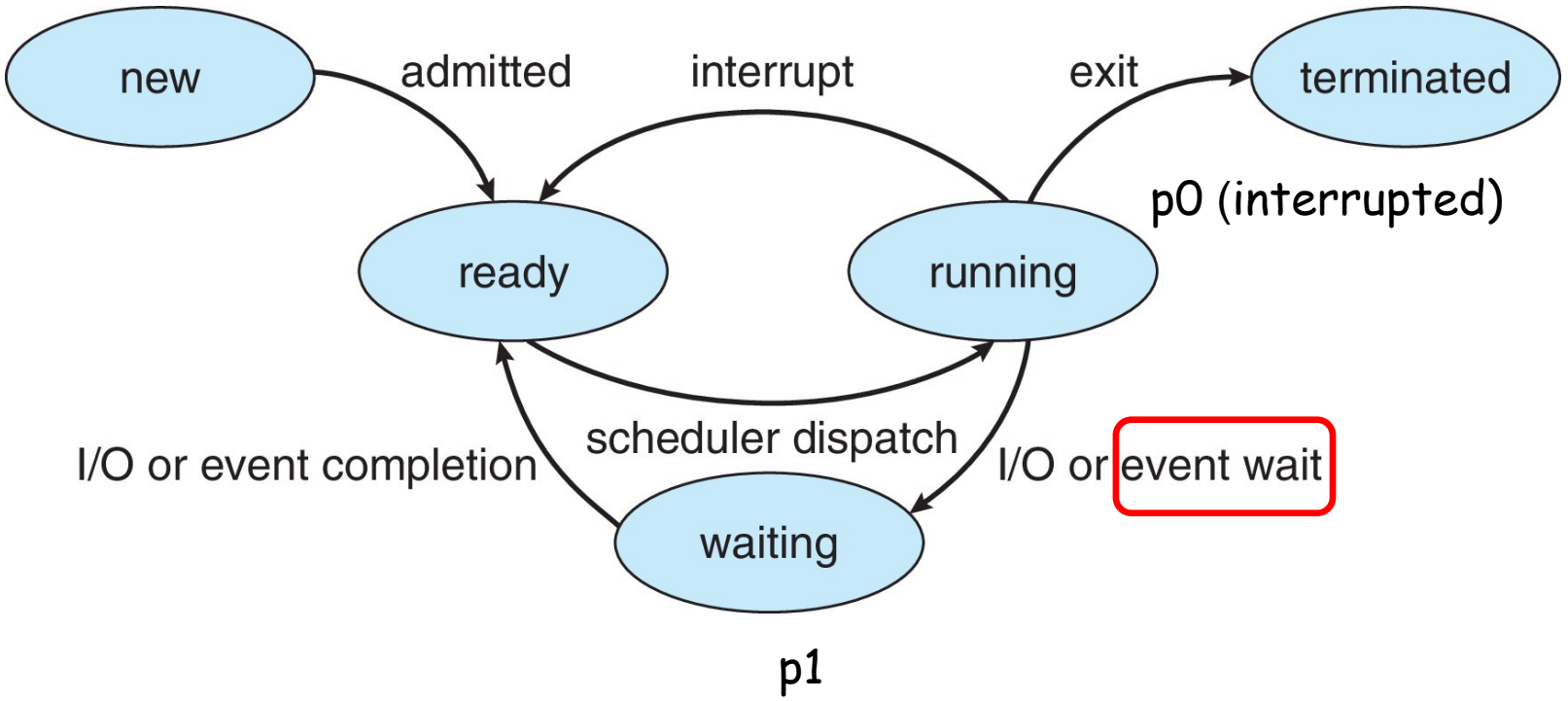
Event: OS dispatch p0



สถานการณ์จำลอง

- I/O controller ส่ง I/O INT มายัง CPU
- CPU รัน I/O Interrupt Handler (โค๊ดของ OS)
  - Save context ของ P0 ไป PCB0
  - (สถานะของ P0 ยังคงเป็น running เนื่องจาก interrupt handler จะขัดจังหวะเพื่อประมวลผลที่ใช้เวลาไม่นาน)

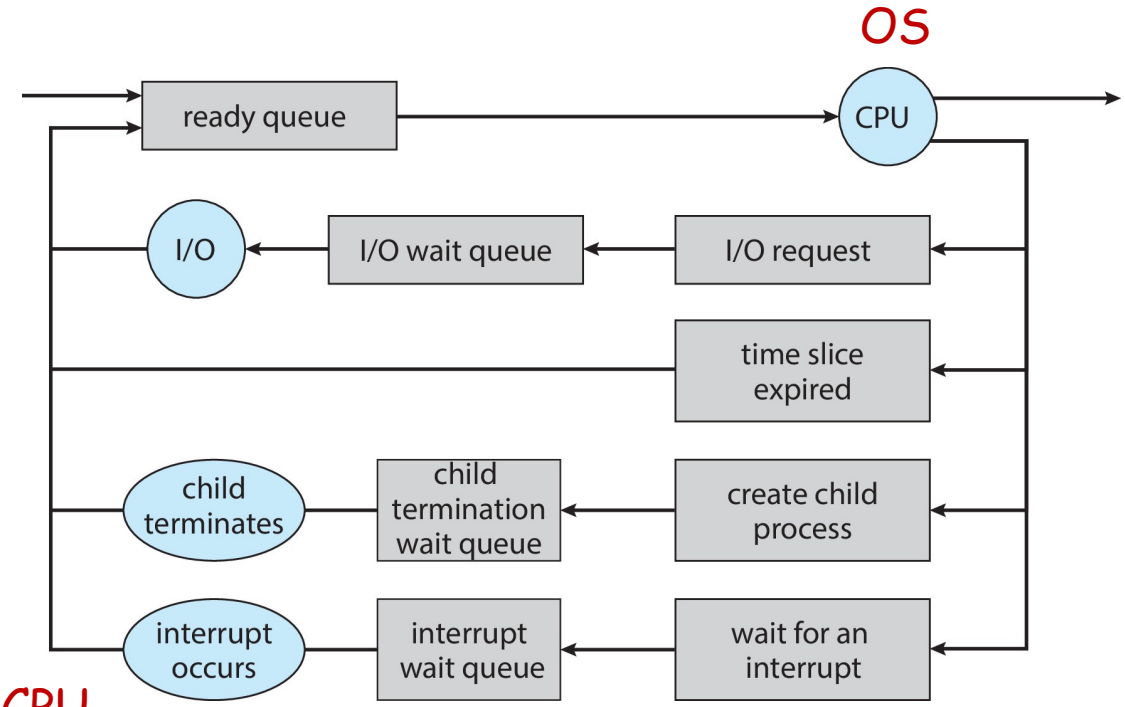
Diagram of Process State



Representation of Process Scheduling

Event: read req of p1 complete and INT

p0 (interrupted)



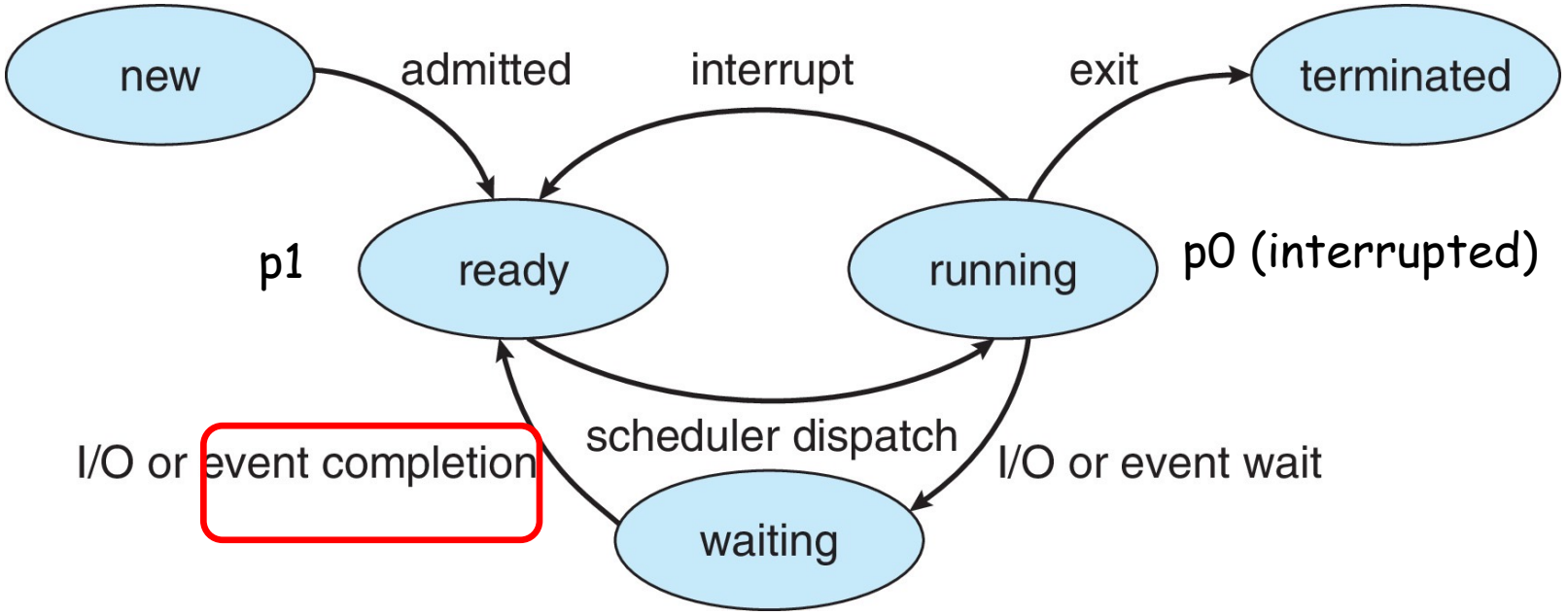
p1

I/O int send to CPU

p0 is interrupted to run INT handler

INT handler is a part of OS

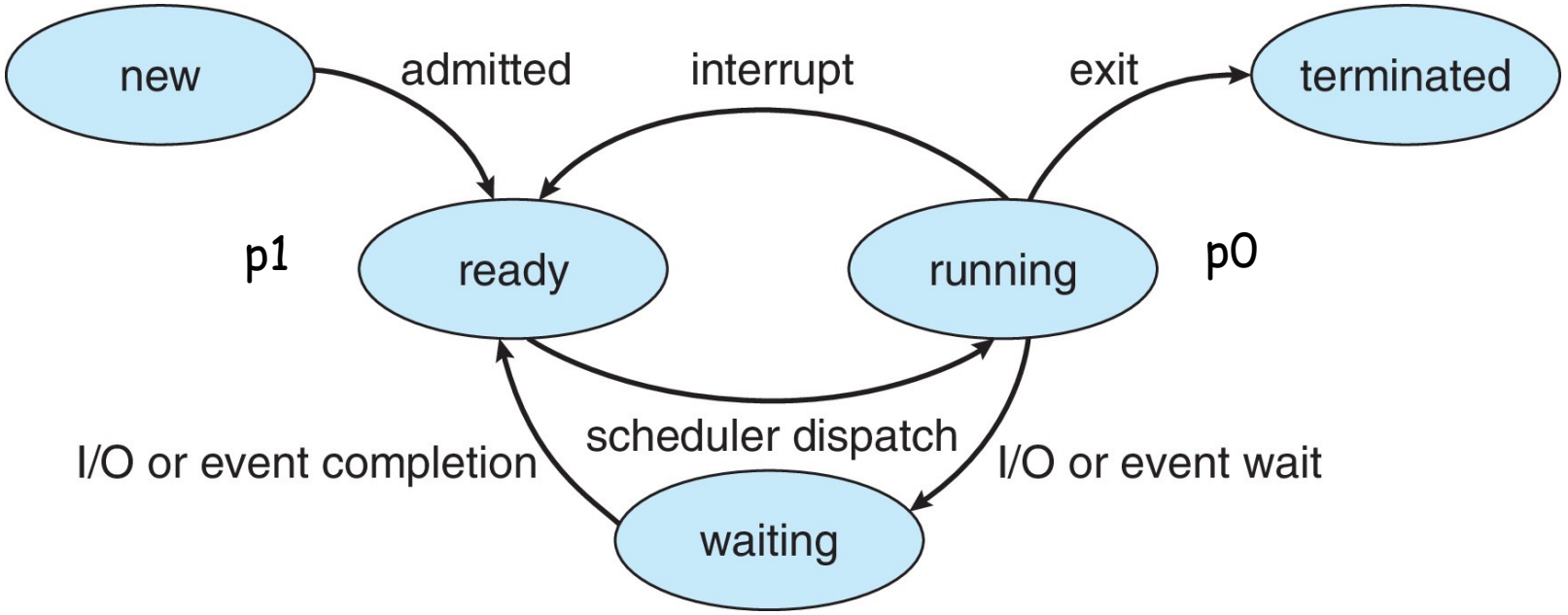
Diagram of Process State



สถานการณ์จำลอง

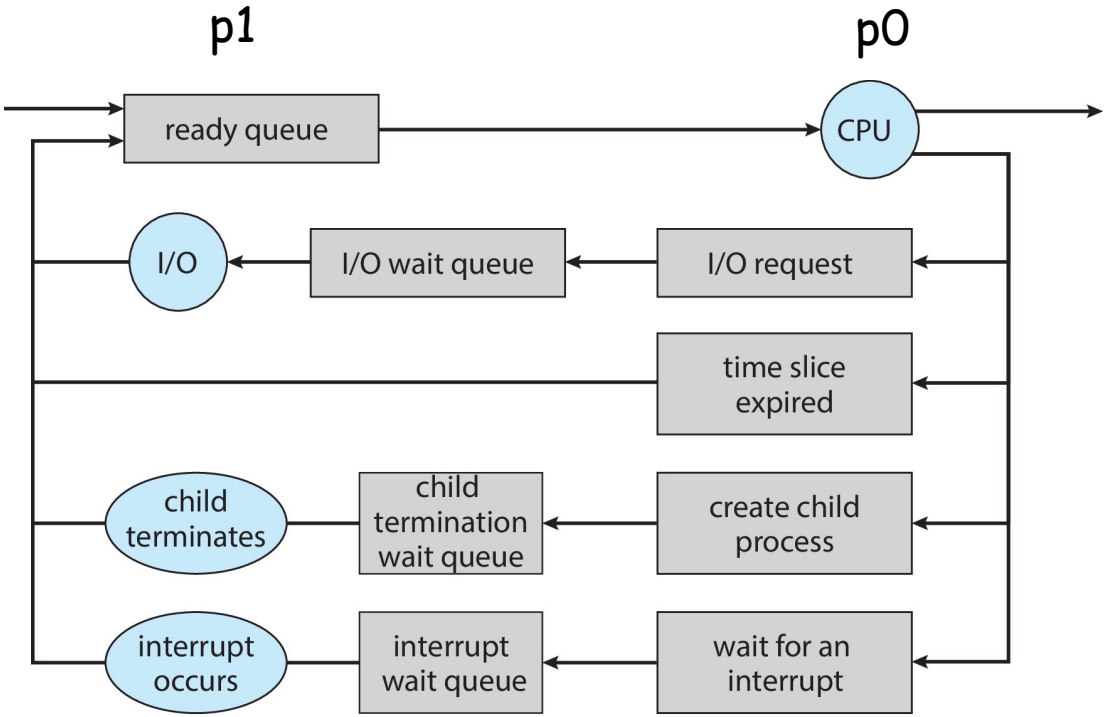
- I/O controller ส่ง I/O INT มายัง CPU
- CPU รัน I/O Interrupt Handler (โค๊ดของ OS)
  - Save context ของ P0 ไป PCB0
  - จัดการ I/O (เช่น เช็คว่า INT เป็นของ Process ใดใน INT wait queue)
  - ลบ p1 จาก INT wait queue
  - นำ p1 ไปไว้ใน ready queue
  - Restore context ของ P0 จาก PCB0 มายัง CPU registers
  - ปล่อยให้ p0 ประมวลผลต่อ

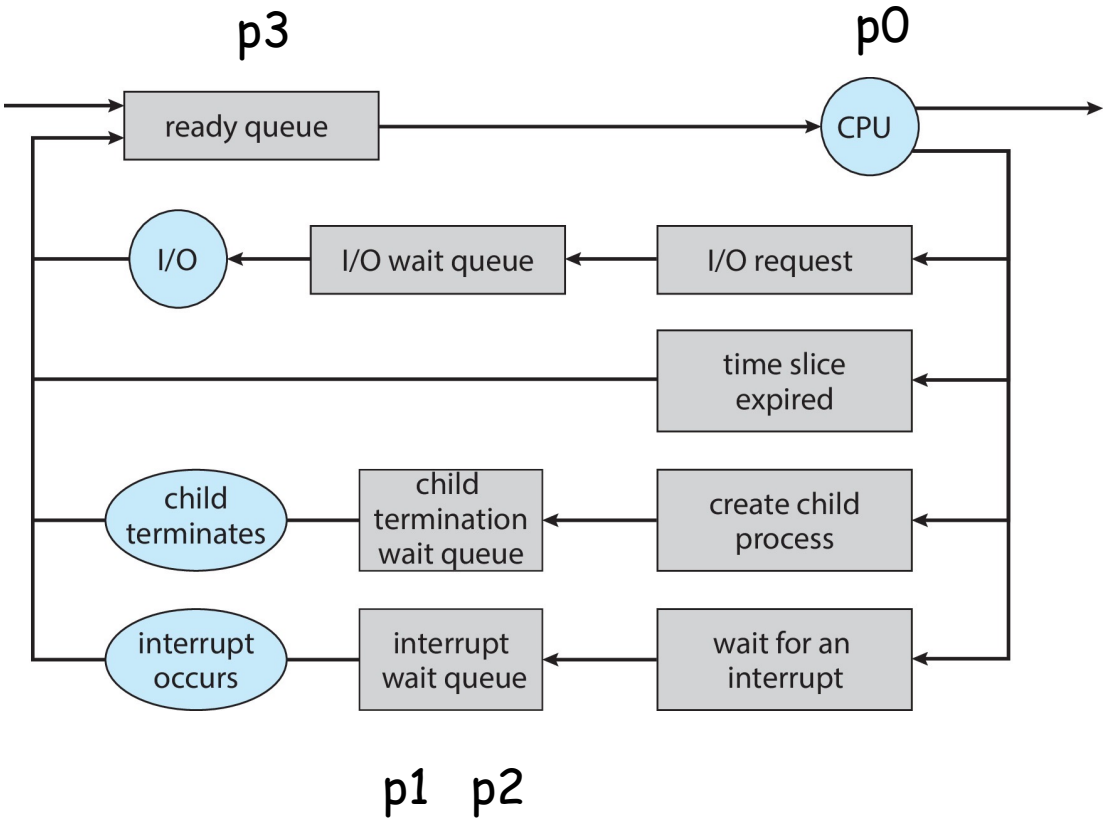
Diagram of Process State

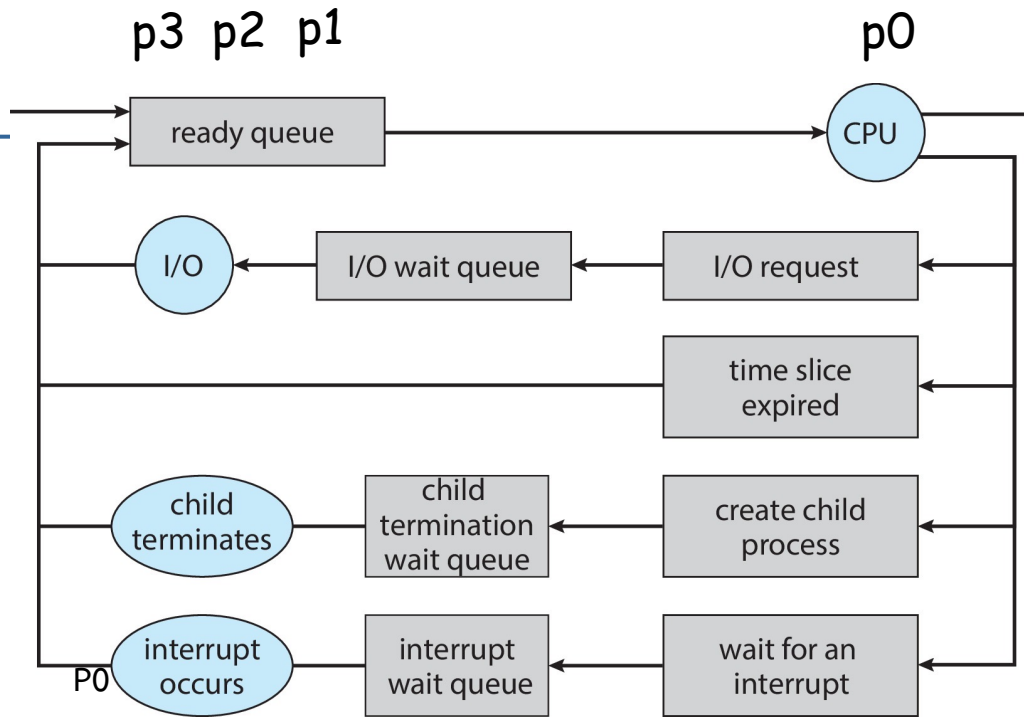


Representation of Process Scheduling

Event: p0 resumes




p3
p0
ready queue
CPU
I/O
I/O wait queue
I/O request
time slice expired
child terminates
child termination wait queue
create child process
interrupt occurs
interrupt wait queue
wait for an interrupt
p1
p2

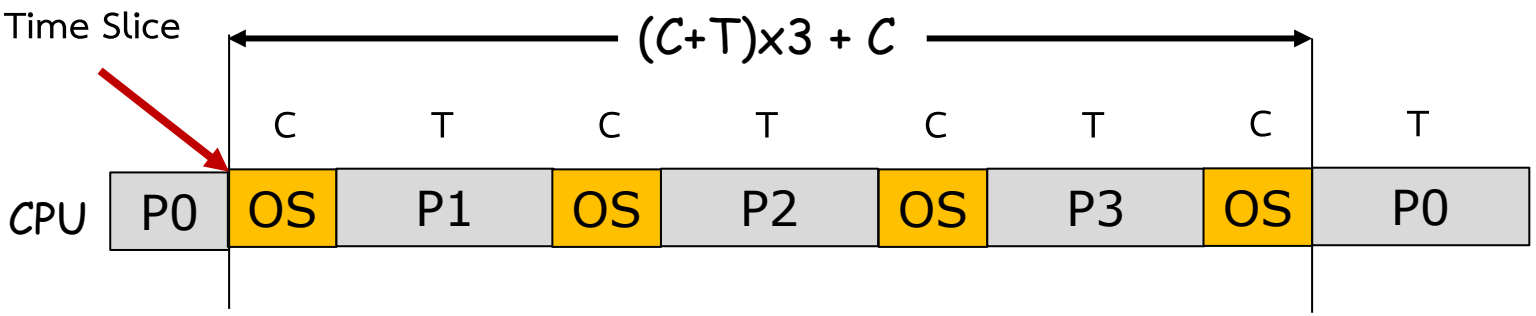


For N+1 processes

$(C+T)xN + C$

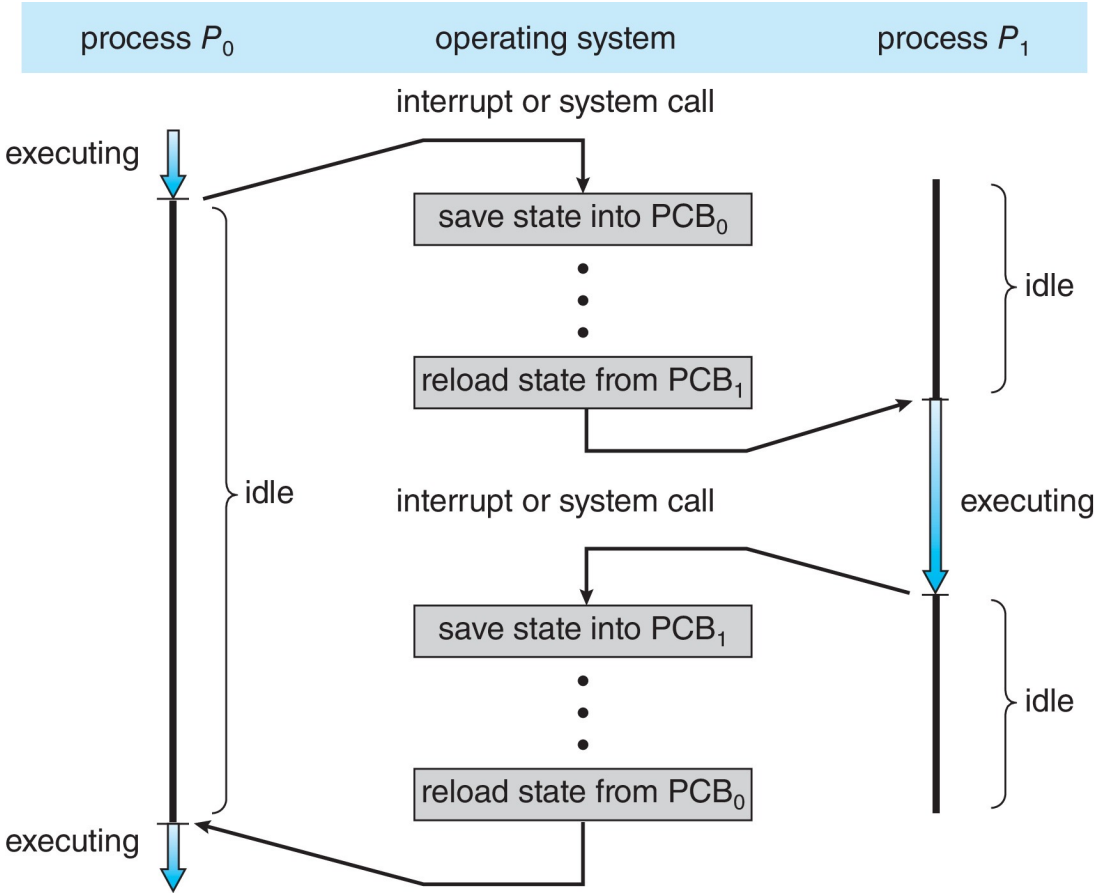
N = 3,
Degree of Multi-programming = N+1 = 4

P0 หมด Time Slice



CPU Switch From Process to Process

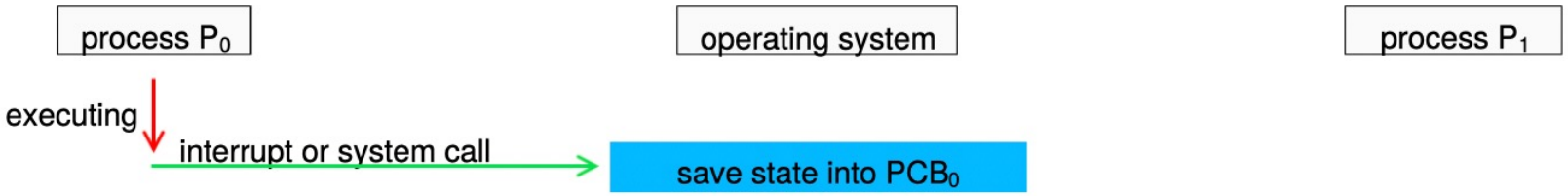
A **context switch** occurs when the CPU switches from one process to another.



Context Switch

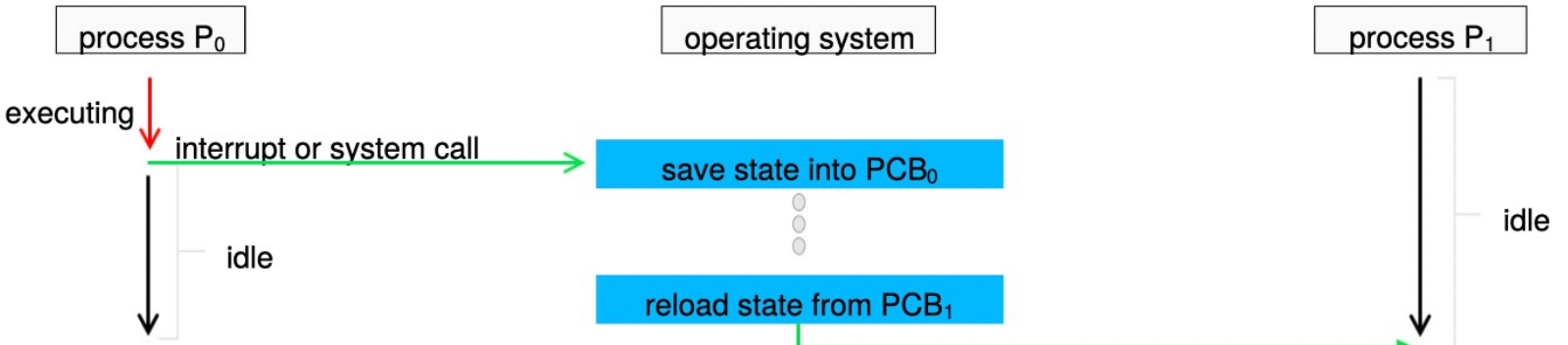
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Context Switch from Process to Process



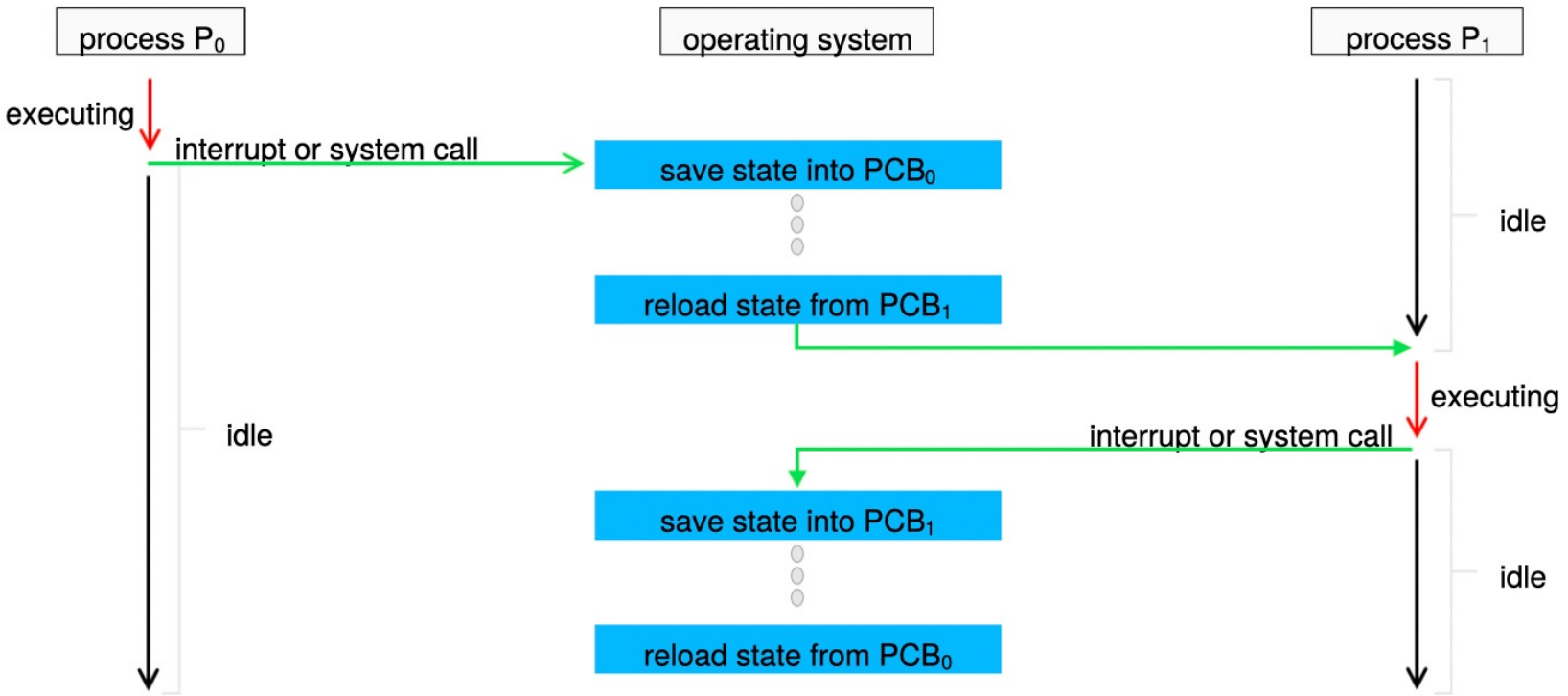
1. the system has two processes, P_0 and P_1
   - P_1 is idle, P_0 is executing and executes a system call, or the system receives an interrupt
   - the operating system saves the state of P_0 in its PCB

Context Switch from Process to Process



2. P_0 is now off core, so idle
   - the operating system restores the state of P_1 from its PCB and puts P_1 on the core
   - it continues execution

Context Switch from Process to Process

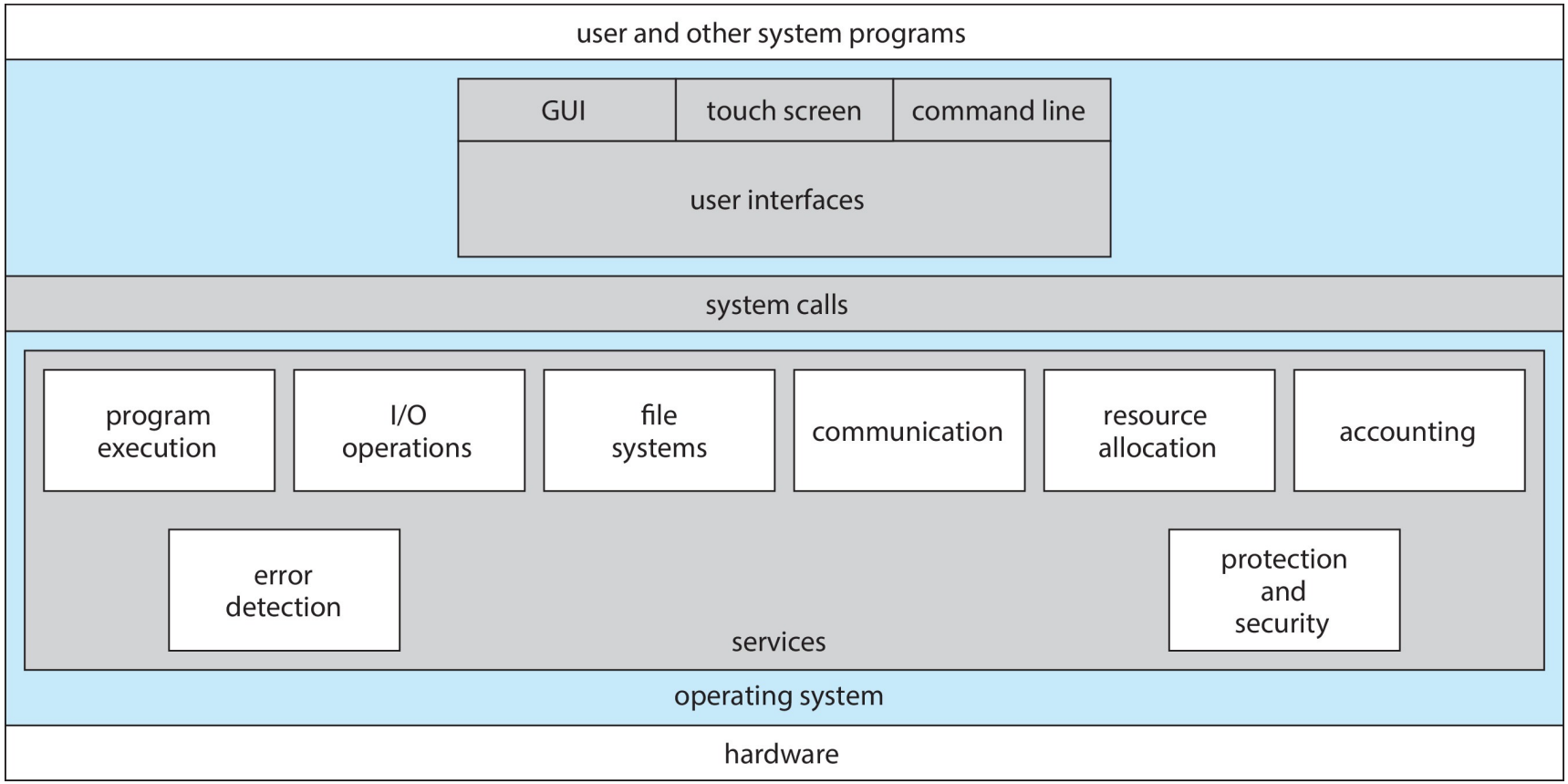


3. P1 execution is interrupted, the operating system saves its state to its PCB and restores the next process's state (P0 in this case) to prepare it to continue execution

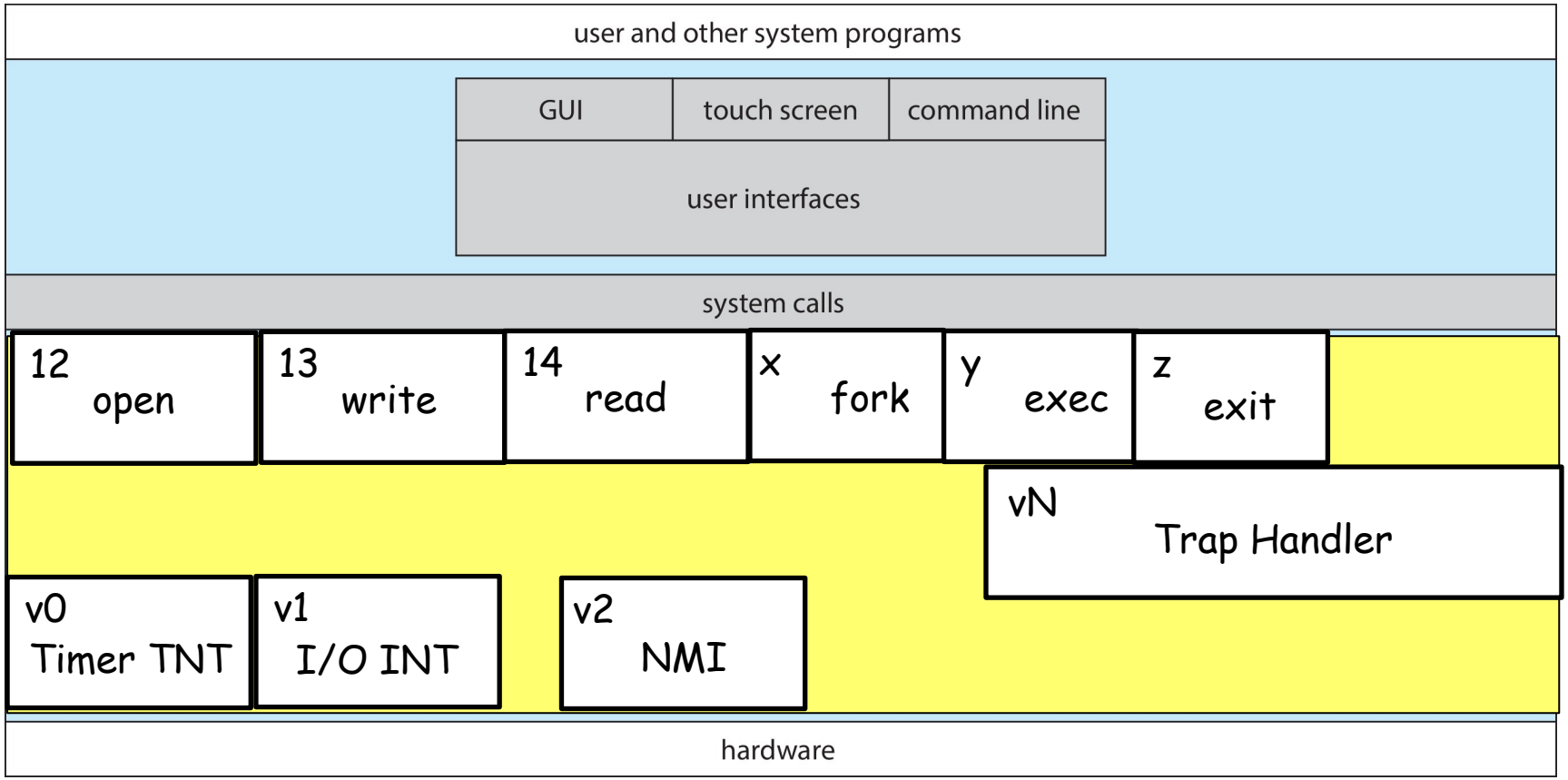
ความเป็นมา

- OS เป็น Interrupt Driven Program
- เมื่อเริ่มต้น boot OS kernel จะถูกโหลดเข้ามาฝังตัวอยู่ในหน่วยความจำ
- เมื่อเกิดอะไรขึ้นในระบบคอมพิวเตอร์ Hardware จะส่งสัญญาณ Interrupt ไปให้ซีพียู
- ทุกครั้งที่ได้รับสัญญาณ Interrupt ซีพียูจะหยุดการประมวลผลปัจจุบันเพื่อรัน Interrupt Handler
- Interrupt Handler/ Trap Handler เป็น function ใน OS Kernel
- OS รองรับการสั่งงานจาก Application Programs ทาง System Call
- System Call เป็น function ใน OS Kernel

A View of Operating System Services



A View of Operating System Services



ความเป็นมา

- เริ่มต้น ยังไม่มี Process ใด นอกจาก OS ที่ถูกโหลดเข้ามาสู่ซีพียู
- เมื่อ OS เริ่มทำงาน มันคือ Process 0
- ทำหน้าที่โหลดทุกอย่างและกำหนดค่าเริ่มต้นทุกอย่างให้เข้าที่
- แล้วตัวมันเองจะสร้าง Process 1 ขึ้นมาให้เป็นผู้สร้าง Process อื่นทั้งหมด
- หลังจากนั้นมันจะอยู่เฉยๆ (idle) ในระบบ (มันจะถูกเรียกให้มาใช้งานซีพียูเมื่อไม่มี Process ใดทำงาน) (ในบาง OS เช่น BSD UNIX มันจะช่วยดูแลระบบด้วย) (ดูรายละเอียดเพิ่มเติมได้ที่ https://superuser.com/questions/377572/what-is-the-main-purpose-of-the-swapper-process-in-unix)

เปรียบเทียบ

- เปรียบเทียบ Process 0 เหมือนผู้ก่อตั้งบริษัท (เช่น google, docker)
- เมื่อผู้ก่อตั้ง ได้วางรากฐานของบริษัท พร้อมทั้งกฎเกณฑ์แล้วก็จ้างผู้จัดการบริษัท ให้ทำหน้าที่บริหารจัดการ Process 1
- ผู้ก่อตั้งก็พัก และนั่งดูผลกำไร ดูเรื่องการลงทุน
- ผู้จัดการ จ้าง ผู้ช่วยผู้จัดการ และหัวหน้าแผนก
- หัวหน้าแผนกจ้างพนักงานในแผนก

Process ID

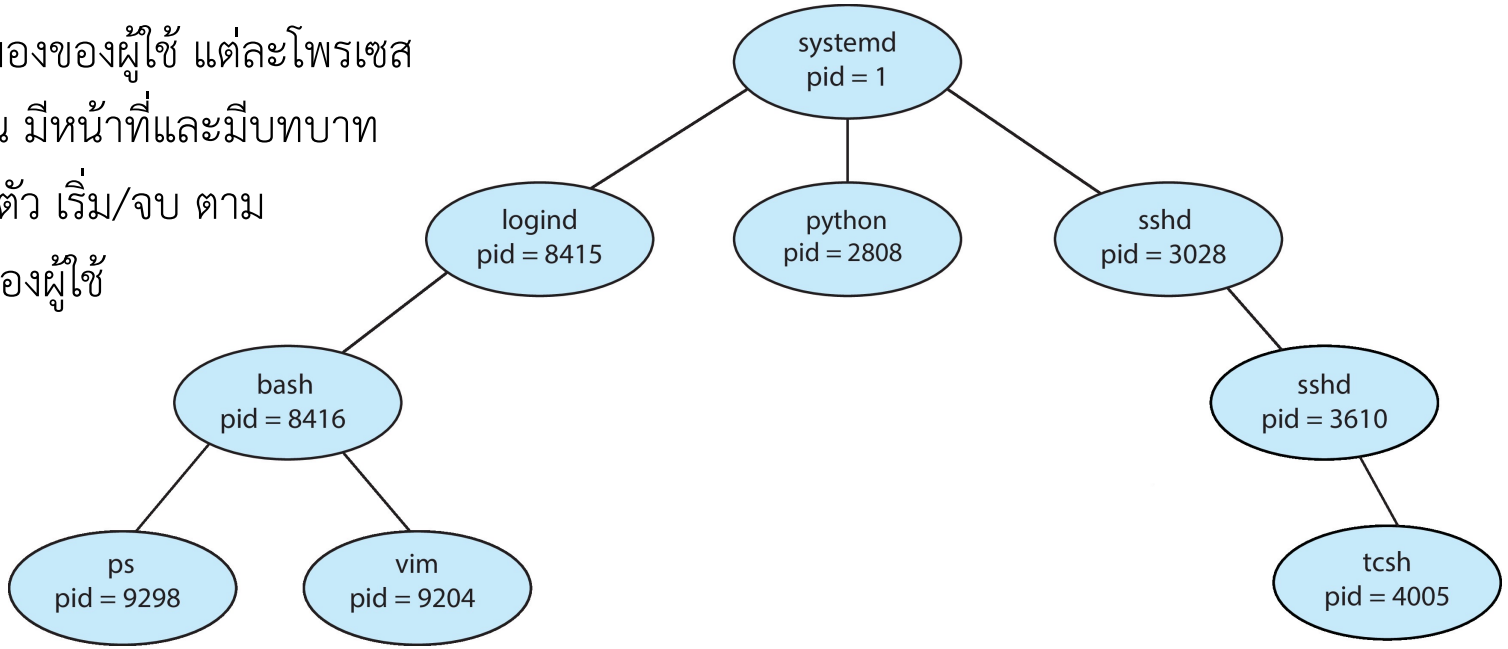
- Process คือโปรแกรมที่กำลังปฏิบัติงาน (และเนื้อหาของโปรแกรมถูกโหลดเข้าสู่หน่วยความจำ)
- ทุก Process มี Process ID
- Process 0 คือโปรเซสแรก เป็นส่วนหนึ่งของ OS kernel เป็น process เดียวที่ไม่มี parent ถูกสร้างขึ้นตอน boot OS
- Process 0 fork() Process 1 เพื่อให้เป็นผู้จัดการ Process อื่นทั้งหมด
- *เมื่อ Process 0 สร้าง 1 เสร็จแล้วมันจะไม่ทำอะไร และ* ***idle***
- *Process 0 จะถูกเรียกเข้ามารันใน CPU เมื่อระบบไม่มีอะไรจะรัน*
- *(Advanced: ใน OS ปัจจุบัน Process 0 อาจมี Thread ช่วยทำงาน)*

Process ID: init/systemd

- Process 1 คือ /etc/init ซึ่ง (ในปัจจุบัน Linux ใช้ systemd)
  - เป็น daemon process ที่รันตั้งแต่เริ่มจนปิดเครื่อง
  - อ่าน /etc/rc* หรือ /etc/init.d/*
  - เปลี่ยนระบบให้รันแบบ multi-user
  - เป็น user process ที่รันด้วย super user privilege และเป็นต้นตระกูลของทุก Process ยกเว้น Process 0 และตัวเอง
  - init จะทำหน้าที่ดูแล orphan process และ zombie ที่ parent process terminate

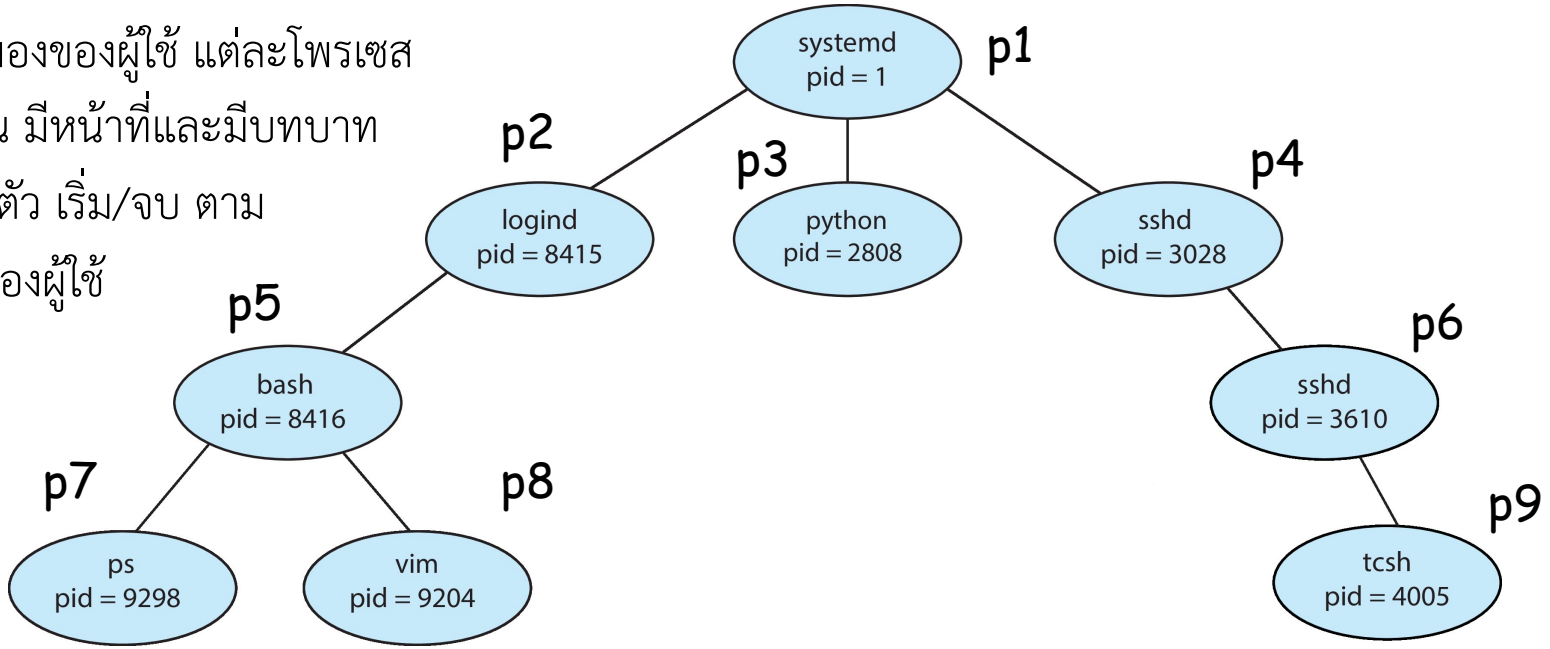
A Tree of Processes in Linux

ในมุมมองของผู้ใช้ แต่ละโพรเซส มีตัวตน มีหน้าที่และมีบทบาทเฉพาะตัว เริ่ม/จบ ตามคำสั่งของผู้ใช้

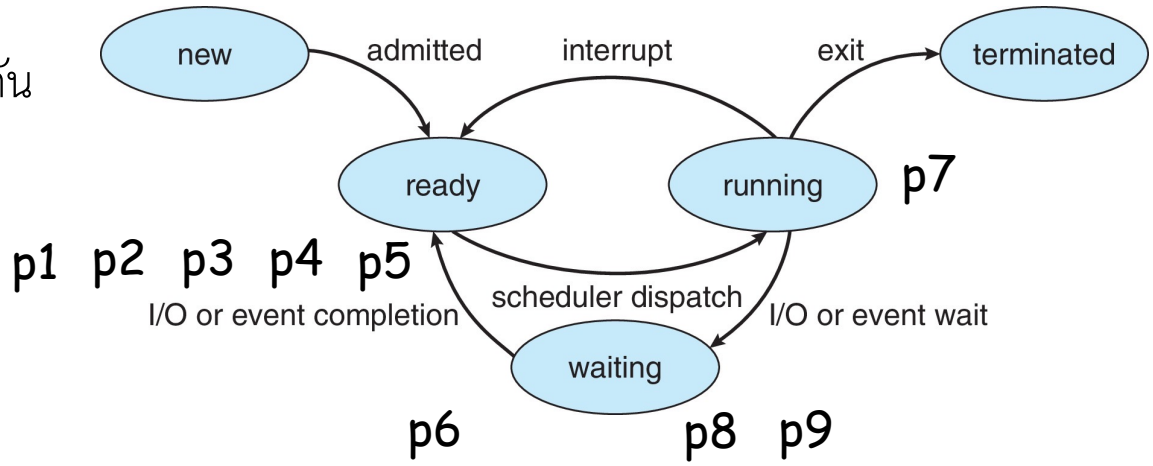


A Tree of Processes in Linux

ในมุมมองของผู้ใช้ แต่ละโพรเซสมีตัวตน มีหน้าที่และมีบทบาทเฉพาะตัว เริ่ม/จบ ตามคำสั่งของผู้ใช้



ในมุมมองของ OS ทุกโพรเซสเท่าเทียมกัน มีบทบาทตามสถานะที่เปลี่ยนไปตามกฎของ OS เท่านั้น



OS Kernel


chapters — u2909610384@cs22201: ~/ch3 — ssh u2909610384@10.100.20.
u2909610384@cs22201:~/ch3$ pstree
systemd─┬─accounts-daemon───2*[{accounts-daemon}]
├─agetty
├─atd
├─cron
├─dbus-daemon
├─dnsmasq───dnsmasq
├─irqbalance───{irqbalance}
├─libvirtd───16*[{libvirtd}]
├─multipathd───6*[{multipathd}]
├─networkd-dispat
├─polkitd───2*[{polkitd}]
├─rsyslogd───3*[{rsyslogd}]
├─snapd───19*[{snapd}]
├─sshd─┬─sshd───sshd───bash
│ ├─sshd───sshd───bash───pstree
│ └─sshd───sshd───bash───sudo───apt───dpkg
├─3*[systemd───(sd-pam)]
├─systemd-journal
├─systemd-logind
├─systemd-machine
├─systemd-network
├─systemd-resolve
├─systemd-timesyn───{systemd-timesyn}
├─systemd-udevd
├─udisksd───4*[{udisksd}]
├─unattended-upgr───{unattended-upgr}
├─upowerd───2*[{upowerd}]
└─uuidd
u2909610384@cs22201:~/ch3$

Operations on Processes

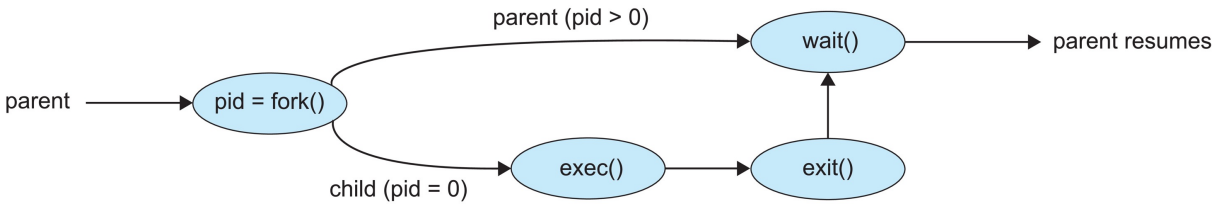
- System must provide mechanisms for:
  - Process creation
  - Process termination

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **`fork()`** system call creates new process
  - **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program
  - Parent process calls **`wait()`** waiting for the child to terminate



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
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - 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

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc., are terminated.
  - The termination is initiated by the operating system.
- 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);
```

- If no parent waiting (did not invoke **`wait()`**) process is a **zombie**
- If parent terminated without invoking **`wait()`**, process is an **orphan**

Wait and waitpid

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or –1 on error

Process Relationship

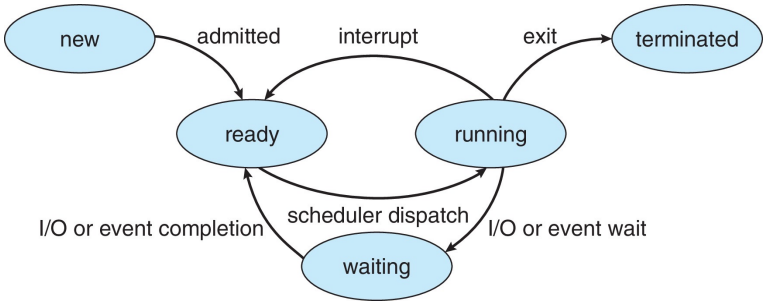
- OS สร้าง Process ที่เป็น System Programs เพื่อให้บริการผู้ใช้
- ผู้ใช้ ใช้ System Programs เพื่อรัน Applications
- Applications เริ่มและจบ ตามความต้องการของผู้ใช้ที่เปลี่ยนแปลงอยู่เสมอ
- เริ่มต้น OS จะเข้ามาเป็น Process 0 และฝังตัวอยู่ถาวรในพื้นที่ส่วนหนึ่งใน Memory ของเครื่องคอมพิวเตอร์
- OS จะให้บริการ System Calls
- Process 0 จะเรียกใช้ fork() system call เพื่อสร้าง Process 1 (ลูก)
- แล้ว Process 0 ก็จะไม่ทำอะไร รอให้ Process 1 จบการทำงาน
- Process 1 จะเรียก exec() system call เพื่อโหลดโปรแกรม init หรือsystemd มาเป็นตัวมัน แล้วเริ่มต้นการประมวลผลของโปรแกรมนั้น

Terminal login

p1

OS

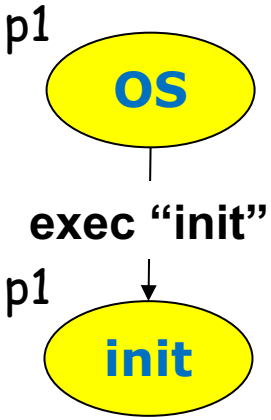
fork

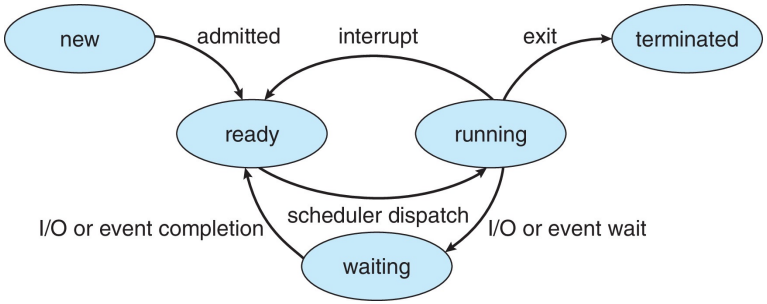


p0

OS Kernel

Terminal login





p0

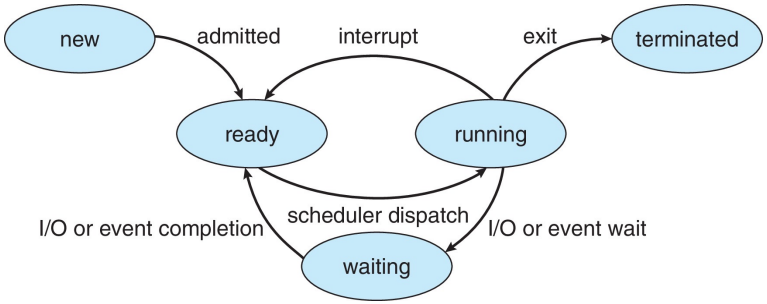
OS Kernel

Terminal login

p1

init

fork/exec

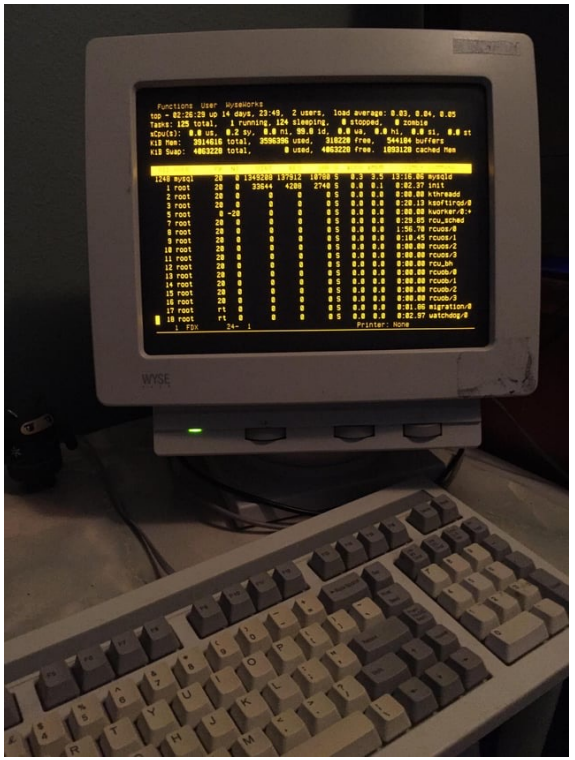


p0

OS Kernel

UNIX Terminal

- Terminal เป็นฮาร์ดแวร์สำหรับแสดงผลอย่างเดียว ไม่มีซีพียู รับข้อมูลจาก mainframe/minicomputer



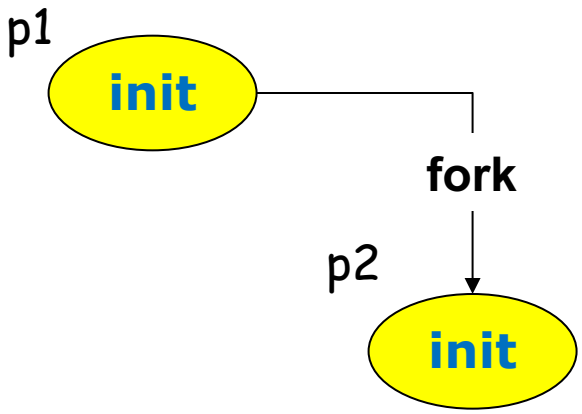
https://web.bii.a-star.edu.sg/systems/faq-sunray.shtml

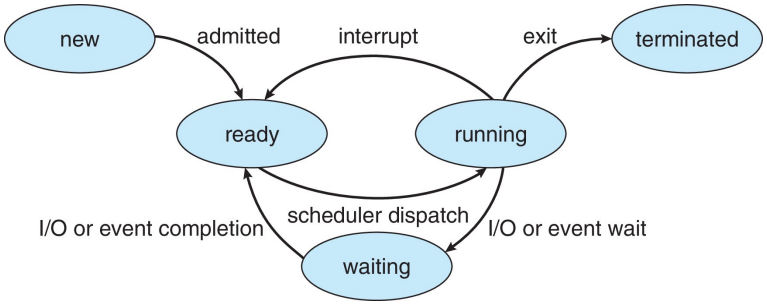
https://www.reddit.com/r/retrobattlestations/comments/45ptfj/my_wyse_terminal_in_amber_hooked_up_to_a_modern/

Terminal Login

- system administrator สร้างไฟล์ /etc/ttys ที่มีรายการชื่อและข้อมูลของอุปกรณ์ terminal หนึ่งบรรทัดต่ออุปกรณ์
- แต่ละบรรทัดมีชื่อของอุปกรณ์และพารามิเตอร์ที่จะป้อนให้กับโปรแกรม **getty**
- เมื่อระบบบูท OS kernel จะสร้าง init (process ID 1) และ init จะเปลี่ยนการทำงานของคอมพิวเตอร์ให้เป็นแบบหลายยูเซอร์ (Multi-users)
- init process จะอ่านไฟล์ /etc/ttys แล้วมันจะ fork process ลูกแล้ว exec โปรแกรม **getty** สำหรับรายชื่ออุปกรณ์ในแต่ละบรรทัด และผ่านพารามิเตอร์ข้อมูลในแต่ละบรรทัดให้กับ
- สมมุติว่าระบบคอมพิวเตอร์ของ นศ มี terminal หนึ่ง terminal

Terminal login

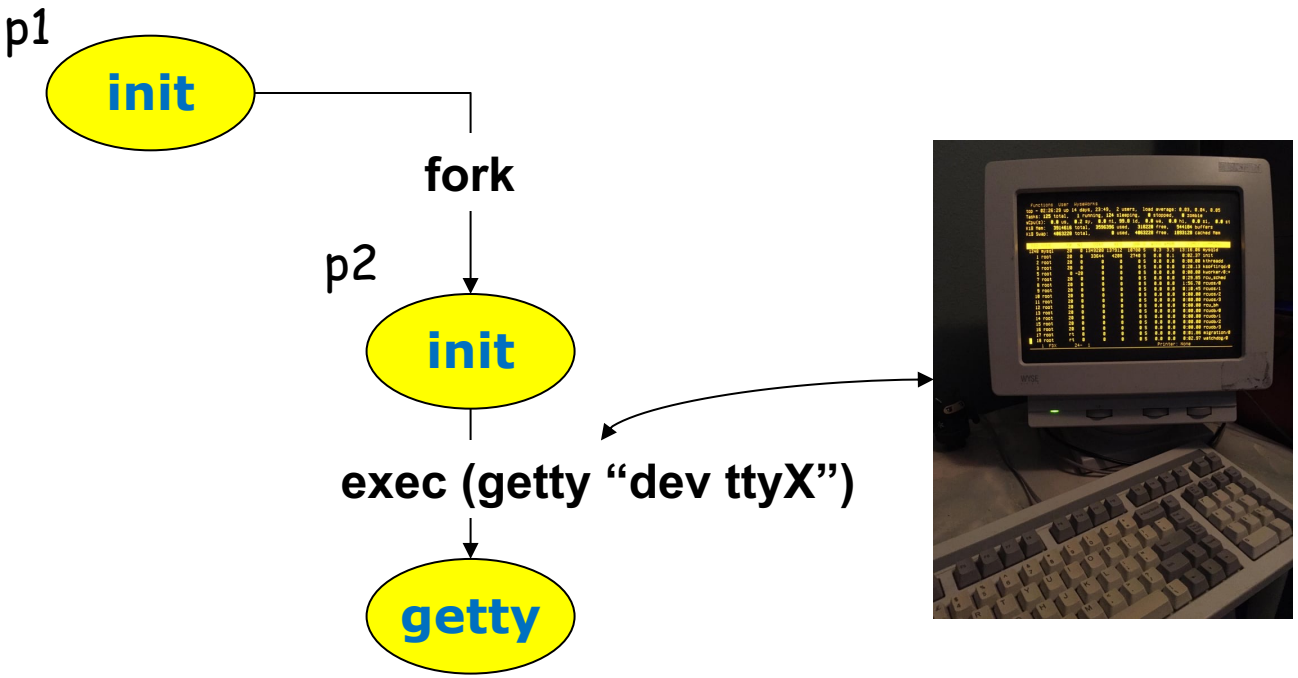


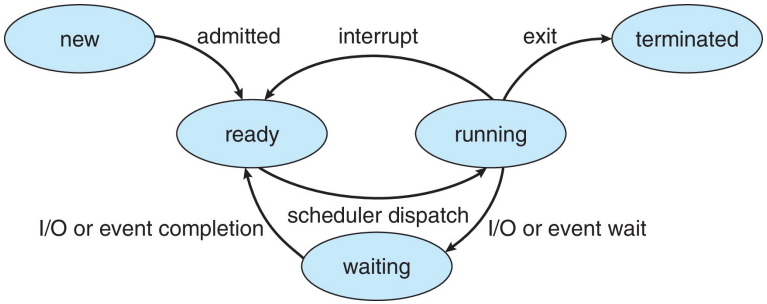


p0

OS Kernel

Terminal login

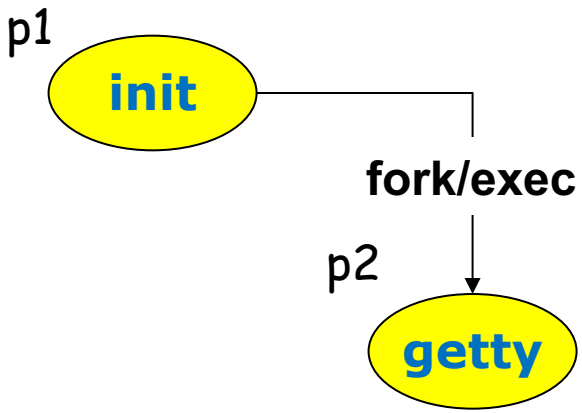


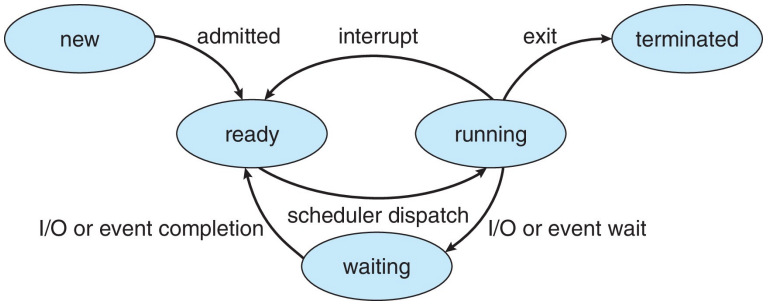


p0

OS Kernel

Terminal login

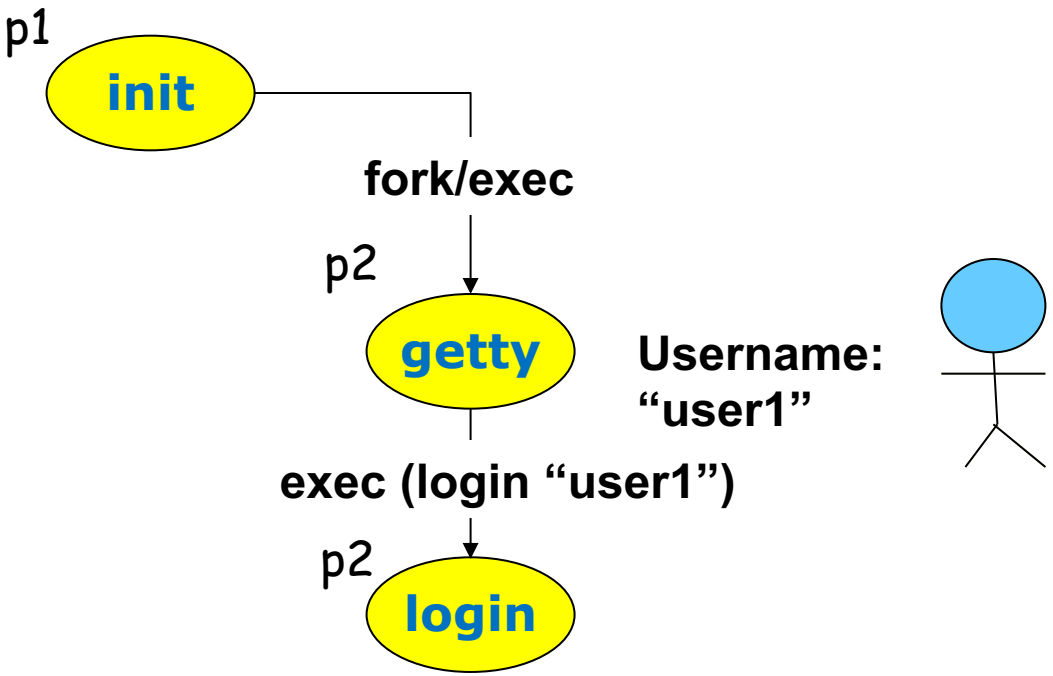


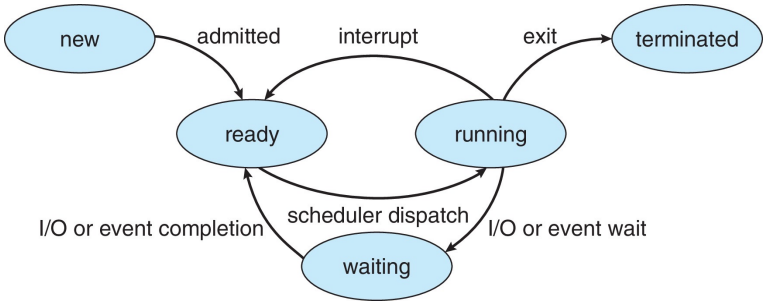


p0

OS Kernel

Terminal login

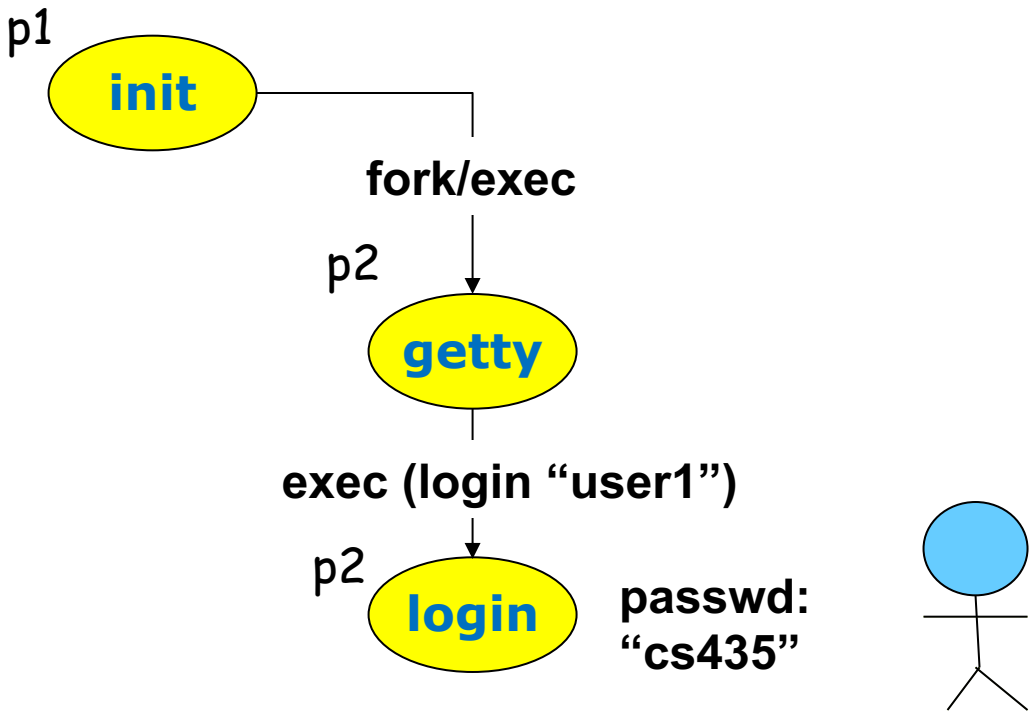


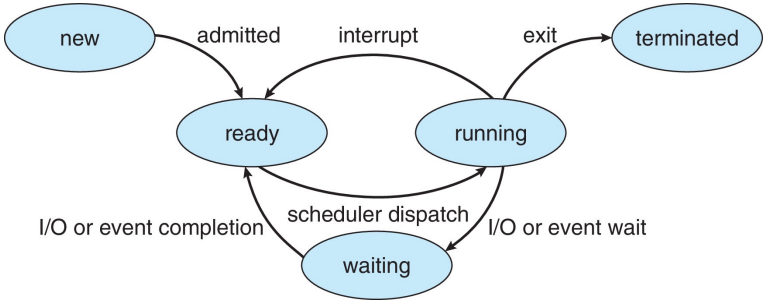


p0

OS Kernel

Terminal login

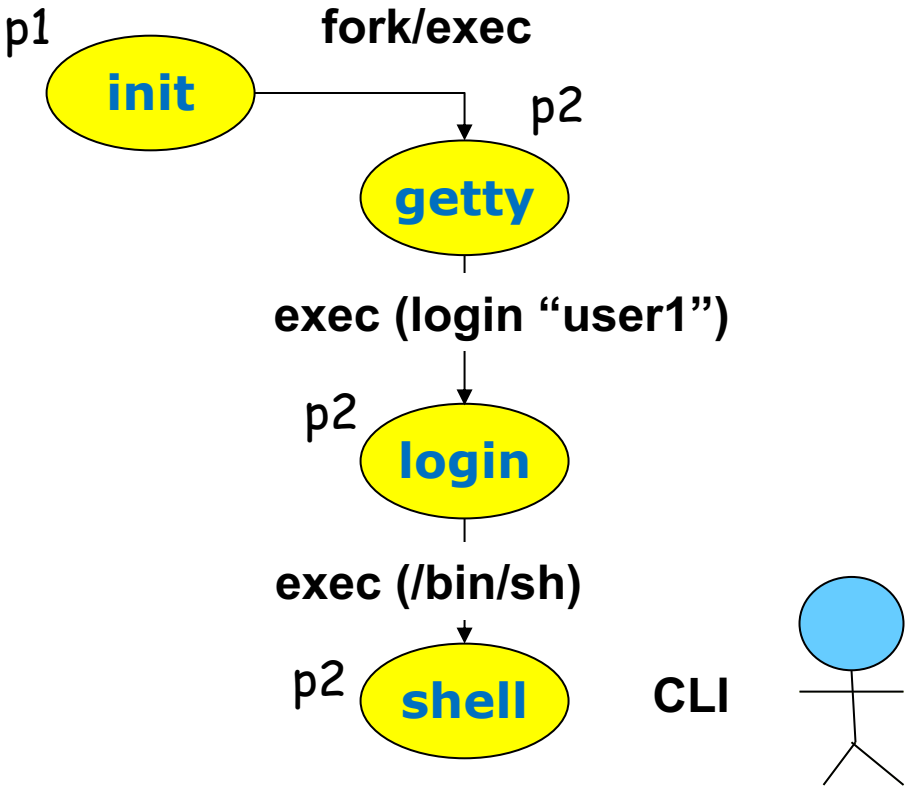


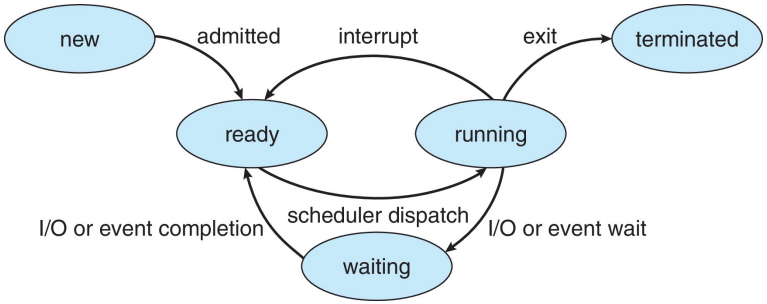


p0

OS Kernel

Terminal login

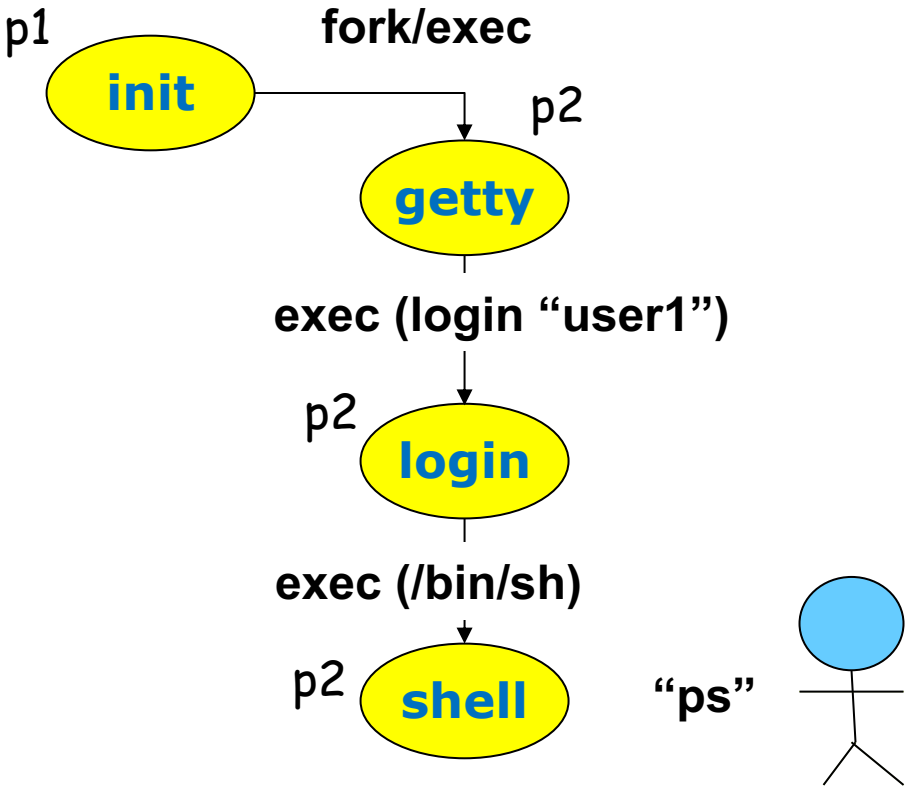


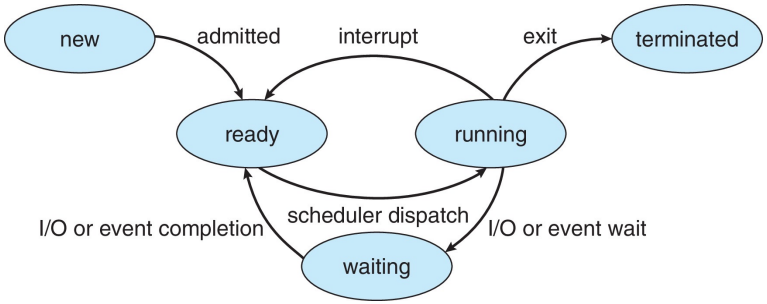


p0

OS Kernel

Run Apps

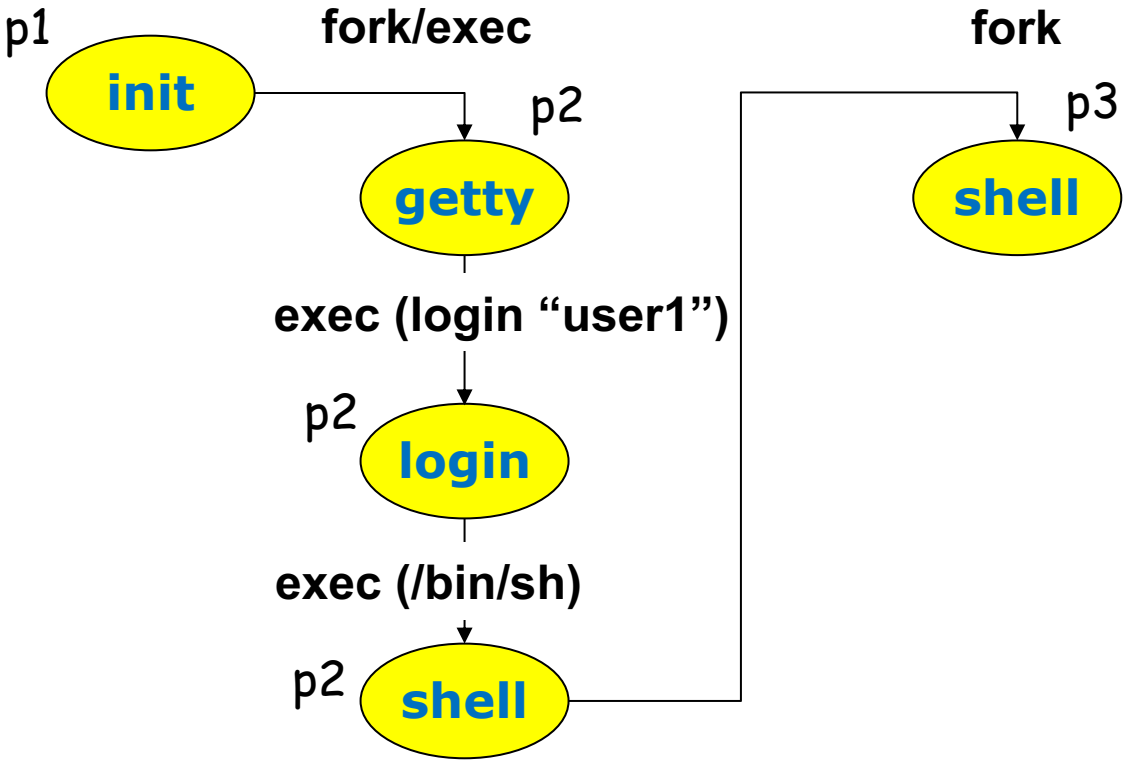


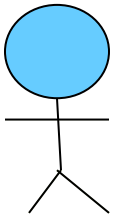


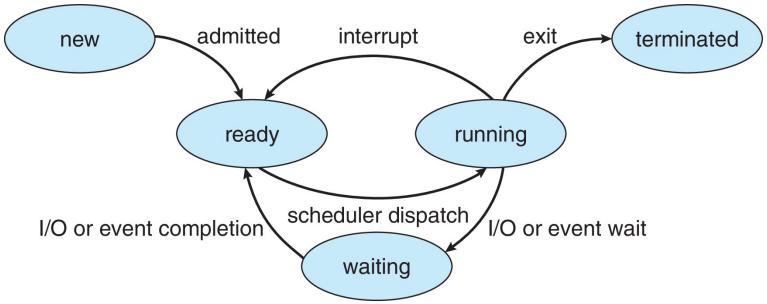
p0

OS Kernel

Run Apps



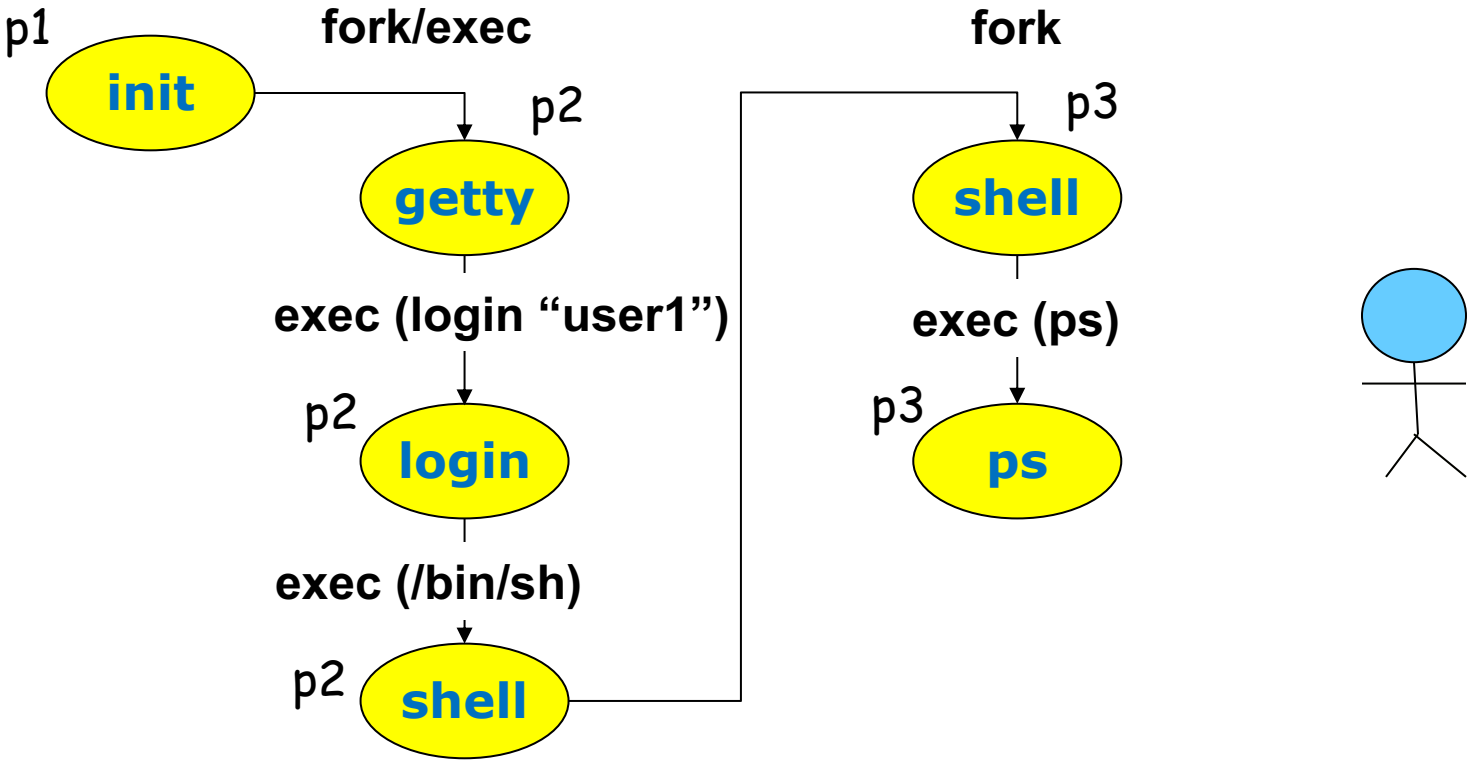


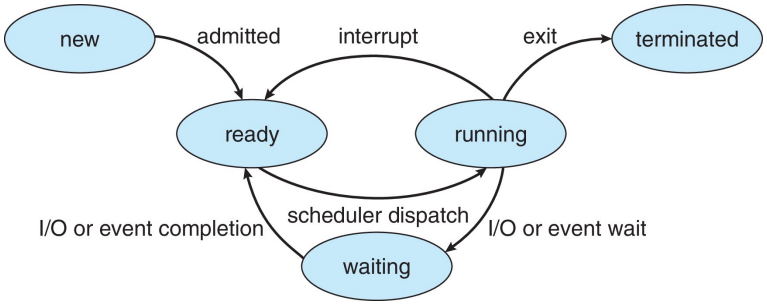


p0

OS Kernel

Run Apps

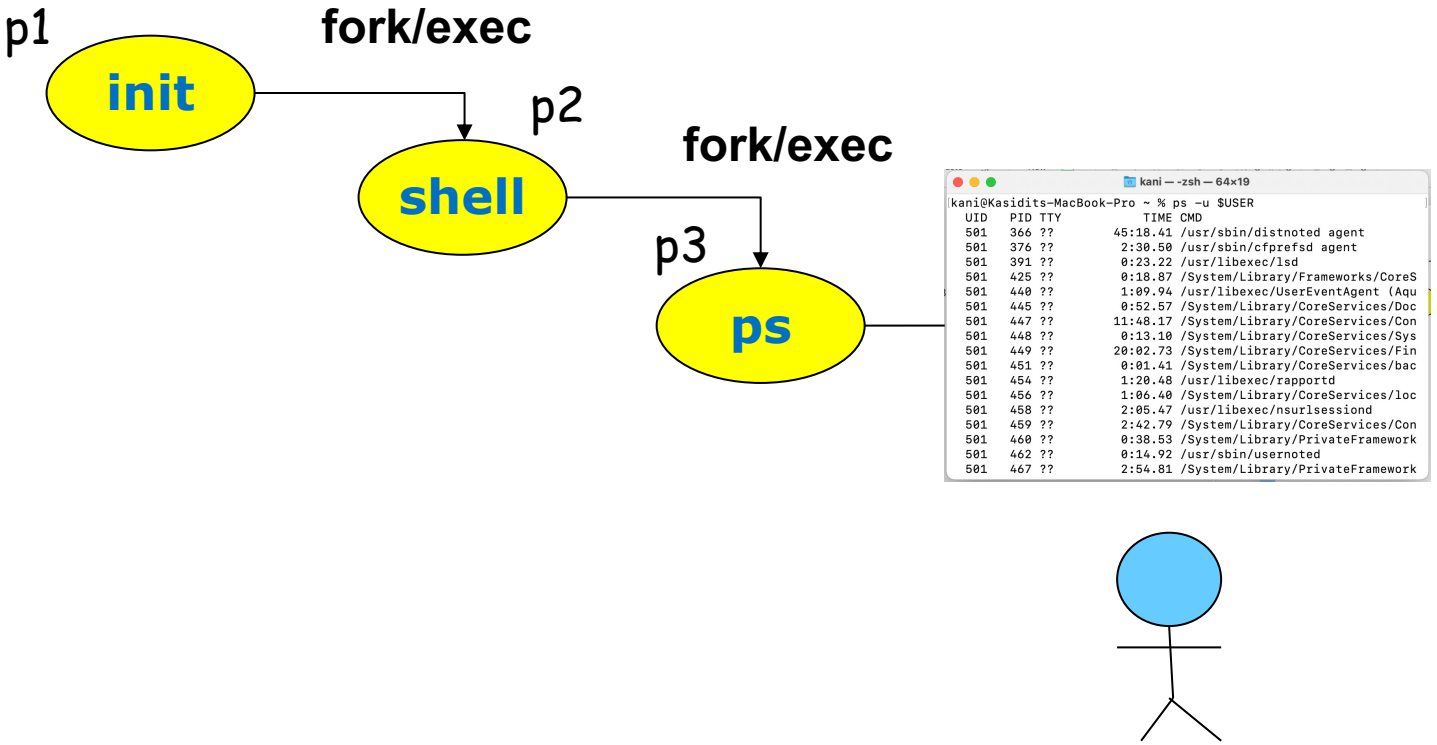


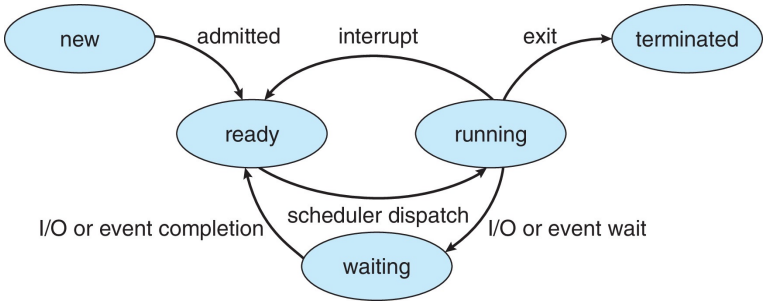


p0

OS Kernel

Run Apps

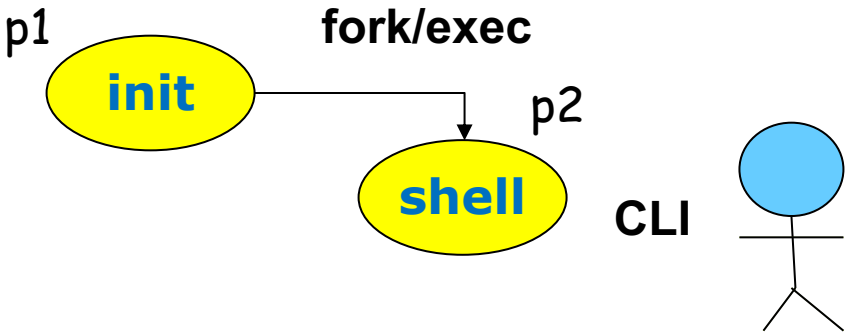


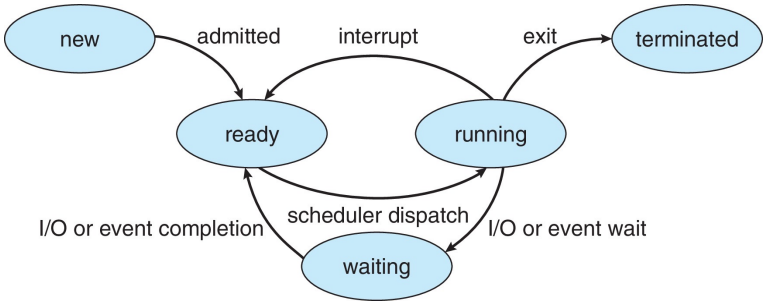


p0

OS Kernel

Run Apps





p0

OS Kernel

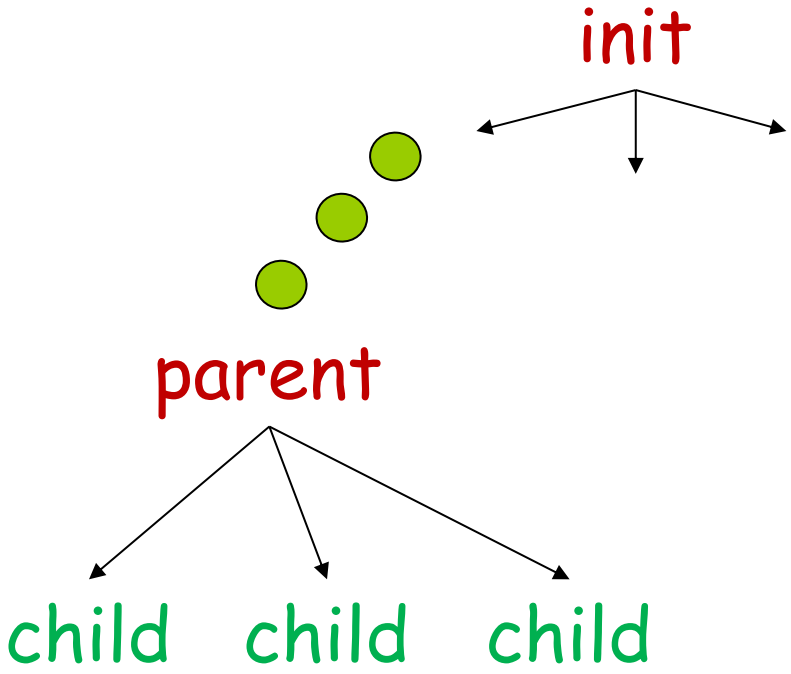
ความสัมพันธ์ระหว่าง Process ที่จบกับ Process อื่น

- เมื่อ Process จบการประมวลผล แล้ว Process อื่นจะต้องทำอย่างไร
- โดยเฉพาะอย่างยิ่ง Process ที่มีความสัมพันธ์กัน
  - ความสัมพันธ์กับ parent
  - ความสัมพันธ์กับ child
  - ความสัมพันธ์กับ init
- การจบของ Process ในรูปแบบต่างๆ มีความสัมพันธ์กับ OS kernel เพราะ OS kernel เป็นผู้จัดการโครงสร้างข้อมูลที่ใช้เก็บ Metadata และค่าสถานะ (State) ของ Process

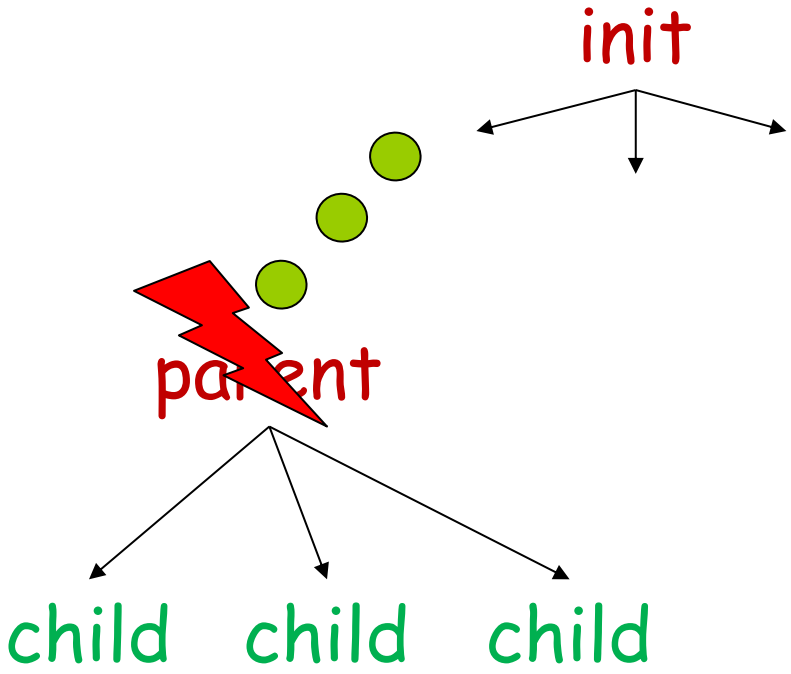
เกี่ยวกับ Orphan Process

- เมื่อ Parent Process จบการทำงานก่อน Child Process
- Child Process จะกลายเป็น Orphan Process
- OS จะกำหนดให้ Orphan Process เป็น ลูกของ Init

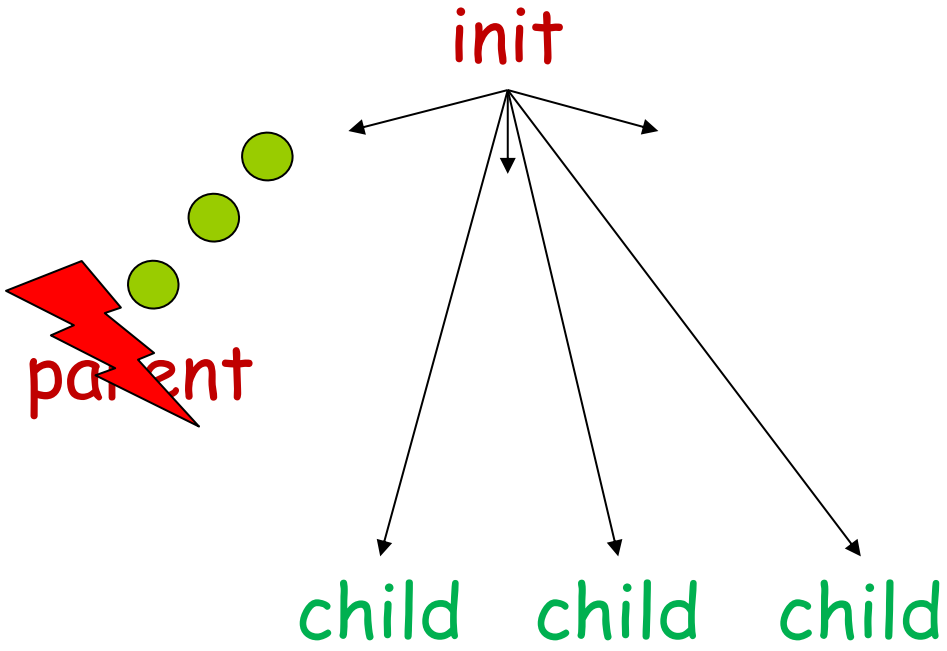
Orphan



Orphan



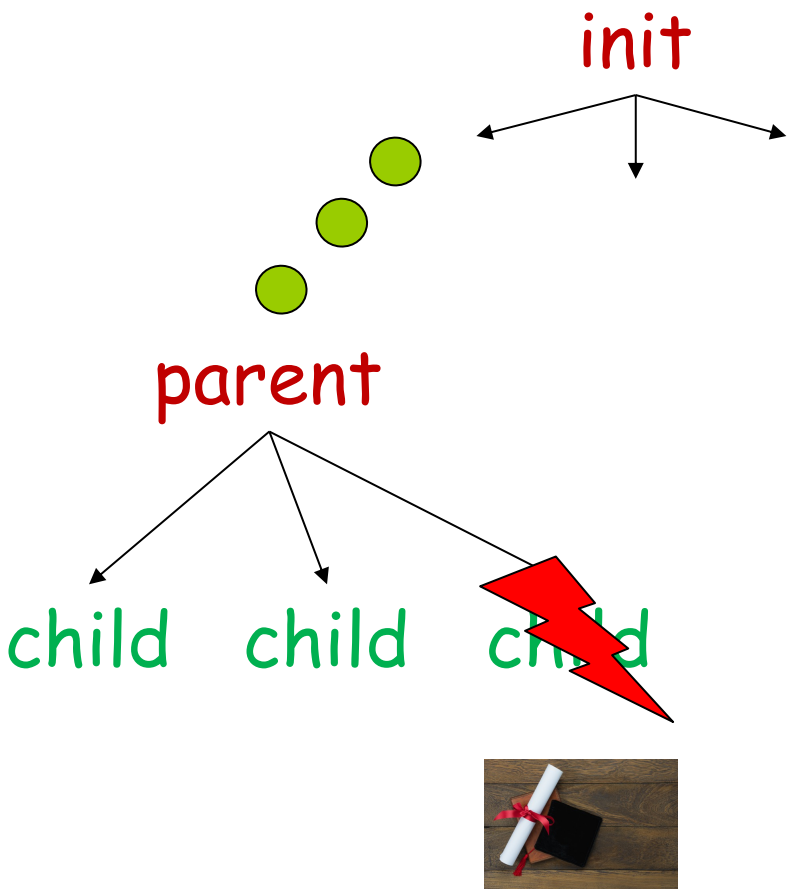
Orphan



Zombie

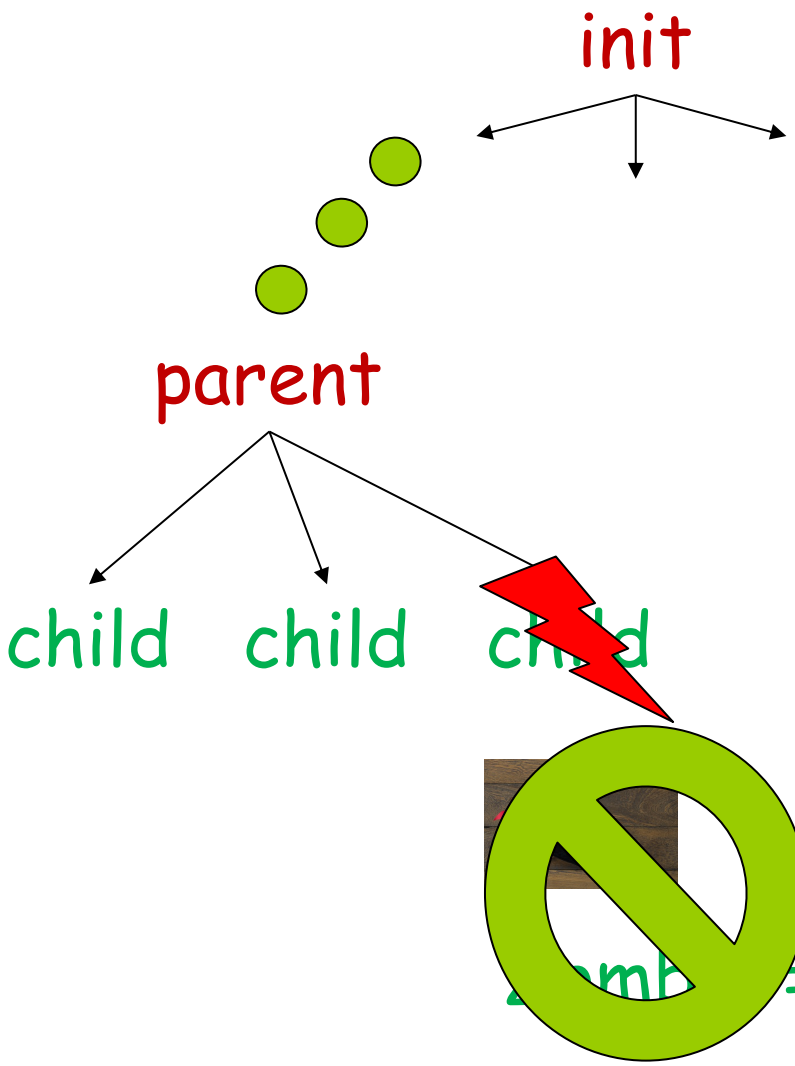
- Child Process จบการทำงานก่อน Parent Process
- เมื่อ Child จบ Parent ควรตรวจสอบสาเหตุการจบทุกครั้ง
- ถ้า Parent Process ไม่ได้มาตรวจสอบสถานการณ์ตายของ Child (โดยใช้ wait system call) ทำให้ Kernel ยังเก็บข้อมูลสถานะนั้นไว้ ข้อมูลนั้นเรียกว่า zombie
- ถ้า Zombie มาก พื้นที่ใน memory ของ kernel ก็จะมาก อาจทำให้ OS ทำงานช้า
- วิธีกำจัด zombie คือต้องให้ parent เรียก wait system call

Zombie



Zombie = ใบมรณบัตร

Zombie



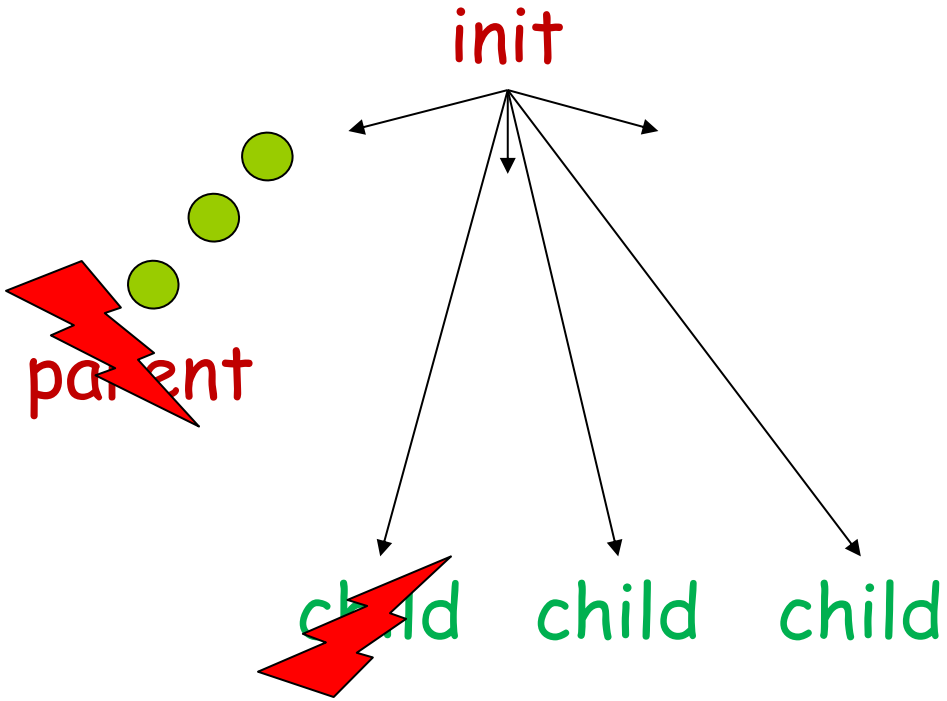
Z ลบไปเมื่อ parent ใช้ wait() เข้ามาดู

= ใบมรณบัตร

Zombie

- ถ้า Child ที่เป็น Orphan จบการทำงาน จะไม่มี Zombie เพราะ OS จะลบข้อมูลสถานะของการจบการทำงาน Zombie ของ Orphan process ทันที

Orphan = No Zombie

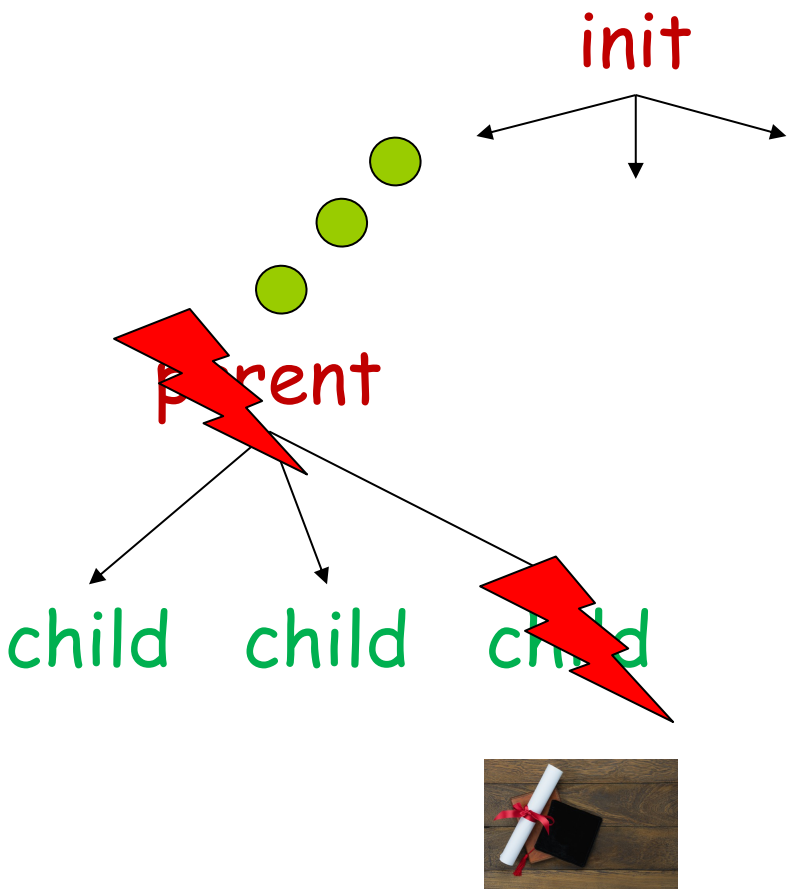


O ตาย init จะ wait() ทันที

Zombie

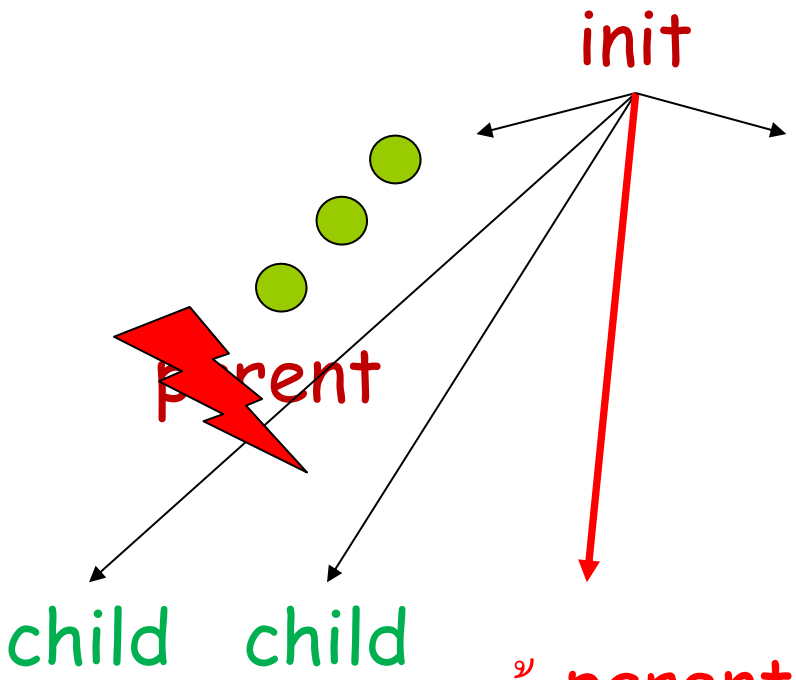
- ถ้า parent process จบการทำงานก่อน เรียก wait system call จะเกิดอะไรขึ้นกับ Zombie
- คำตอบ: INIT จะลบสถานะของการจบการทำงาน Zombie ทันที

Zombie



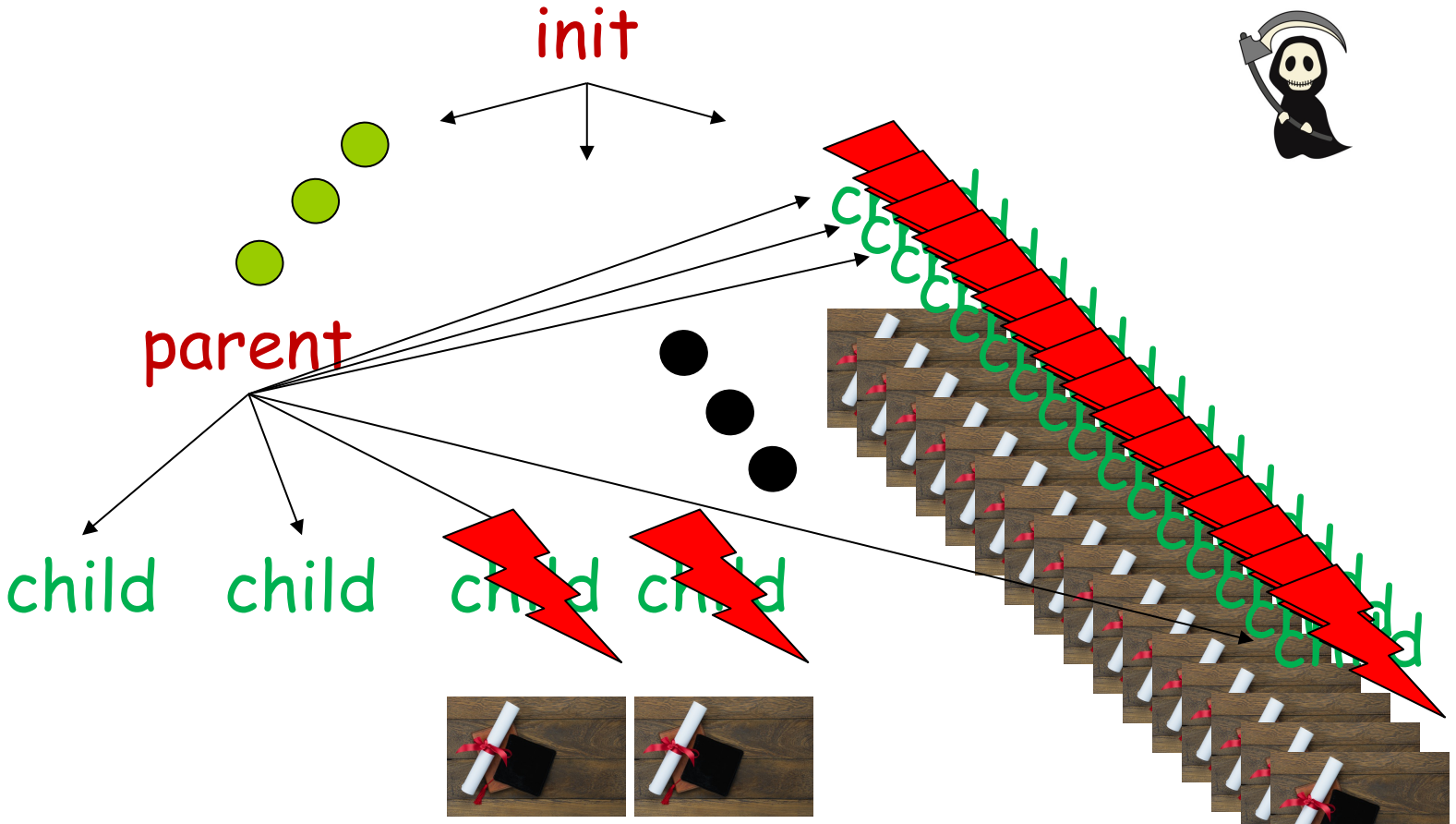
Zombie = ใบมรณบัตร

Zombie



ถ้า parent ตาย
init จะลบ Zombie ให้

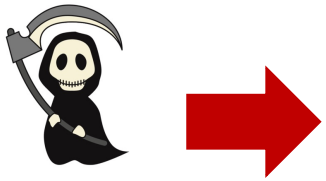
คำถาม How to get rid of Zombie processes in Linux?



คำถาม How to get rid of Zombie processes in Linux?

- มีสองวิธี
- 1. ถ้าโปรแกรมเมอร์ เขียน parent program ให้ รัน signal handler (ซึ่งเป็นฟังก์ชั่นที่จะขัดจังหวะการทำงานของ parent process) เมื่อ parent process ได้รับสัญญาณ SIGCLD จากผู้ใช้ ในกรณีนี้ โปรแกรมเมอร์อาจเขียนคำสั่งใน signal handler ให้เรียก wait() หรือ waitpid() เพื่อกำจัด zombie
  - ปกติเวลา child จบ OS จะส่ง SIGCLD มาให้ parent อยู่แล้ว
  - ในกรณีที่ parent kill ไม่หมด ผู้ใช้ต้องออกคำสั่ง $kill –s SIGCLD <ppid> เพื่อส่ง signal
- 2. Kill parent process แล้ว INIT จะเข้ามากำจัด zombie ทั้งหมดให้ทันที

1st option

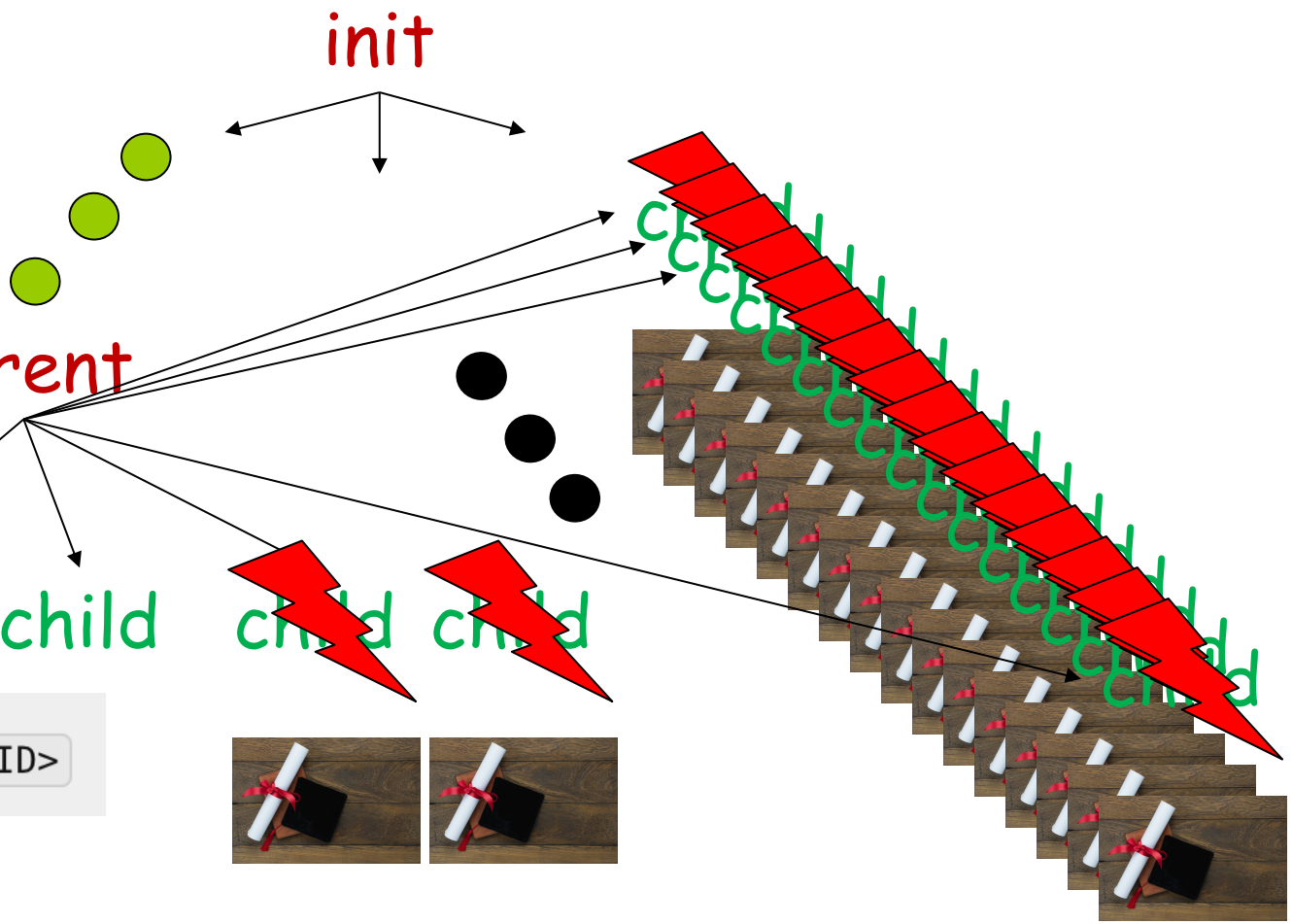


บอกให้

parent

wait()

ลูกที่จบแล้ว




```
kill -s SIGCHLD <PPID>
```

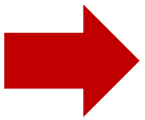
Zombie = ใบมรณบัตร

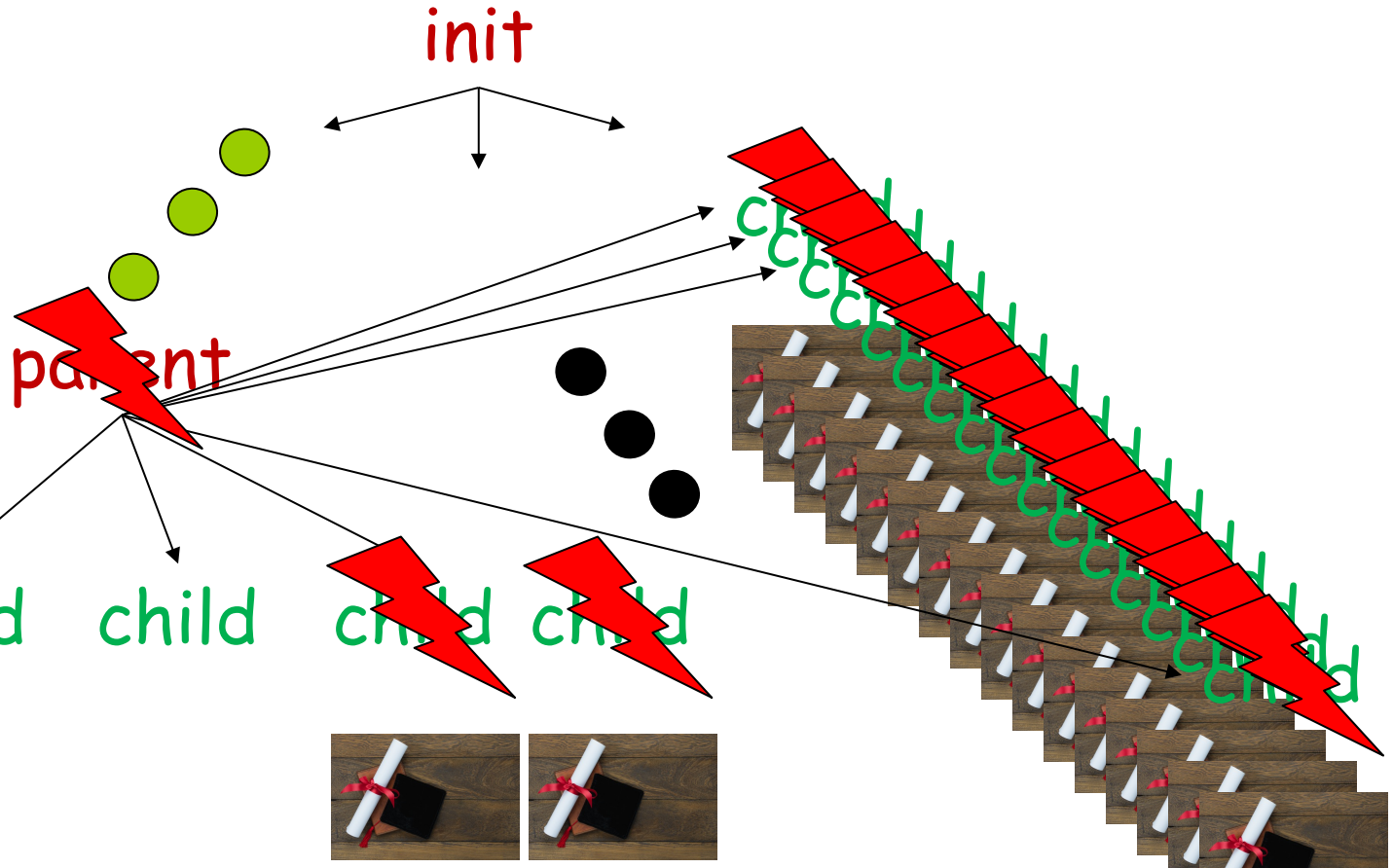
2nd option

```
kill -9 <PPID>
```

กำจัด

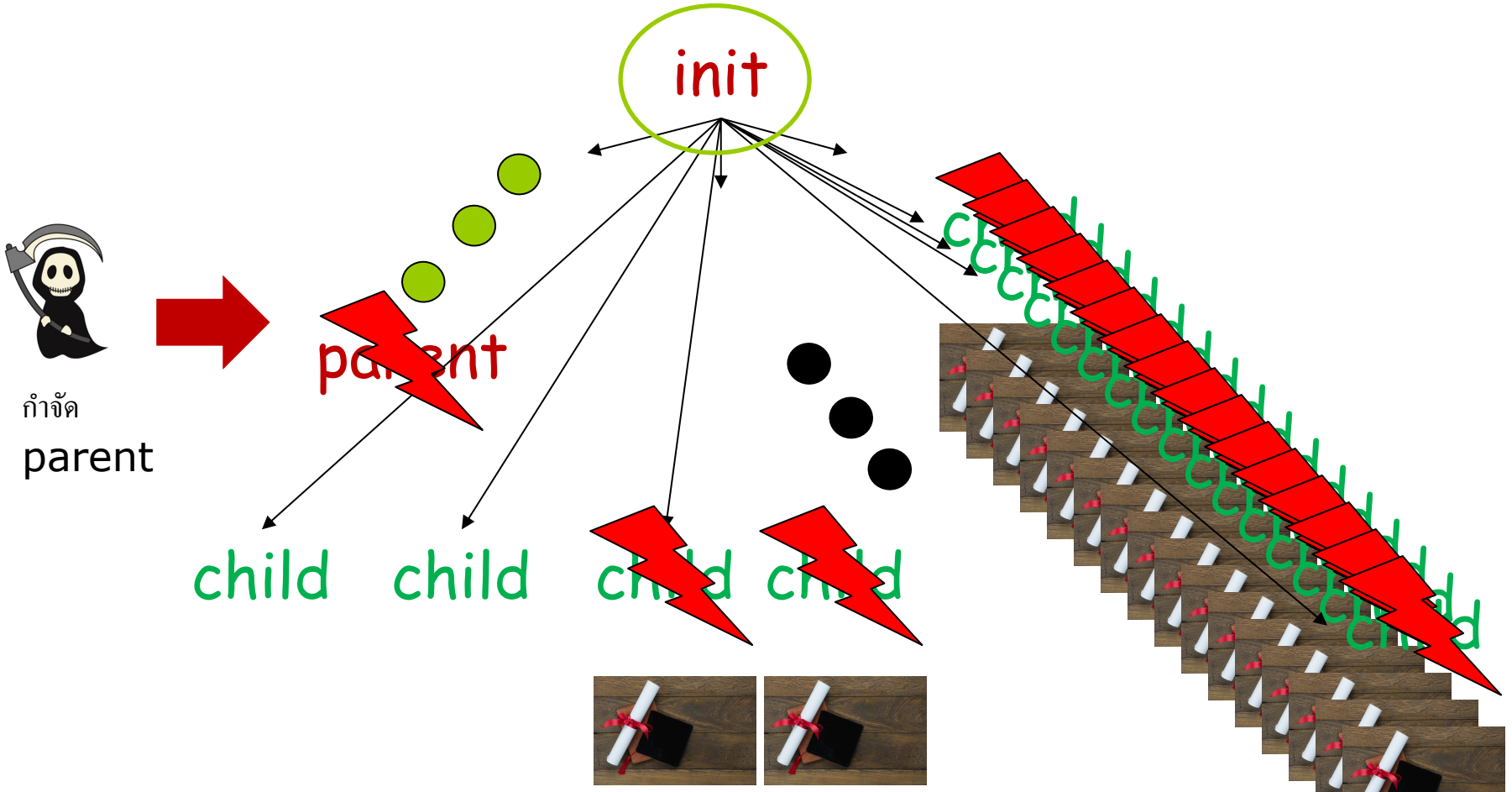
parent





Zombie = ใบมรณบัตร

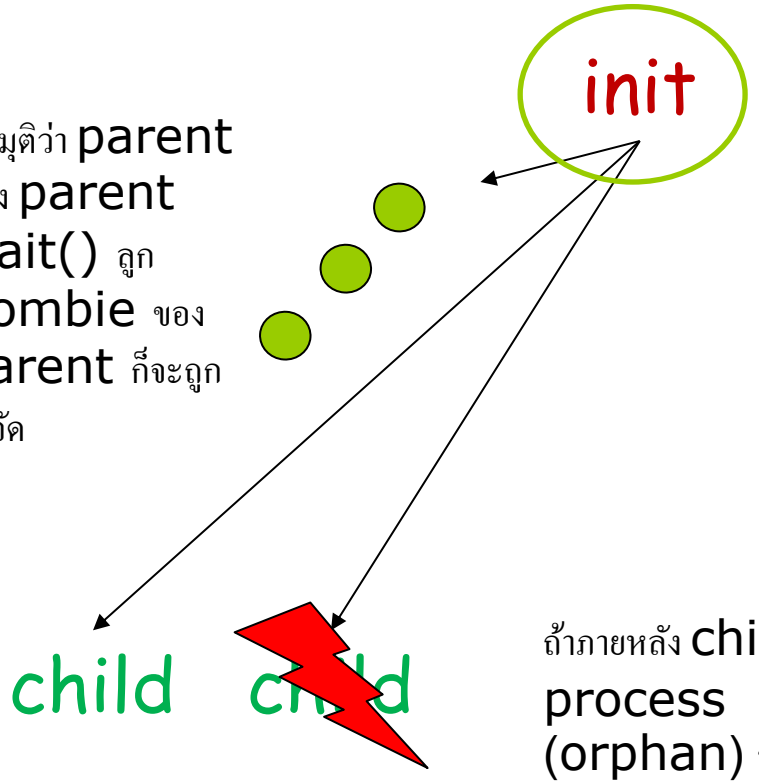
2nd option



Zombie = ใบมรณบัตร

2nd option

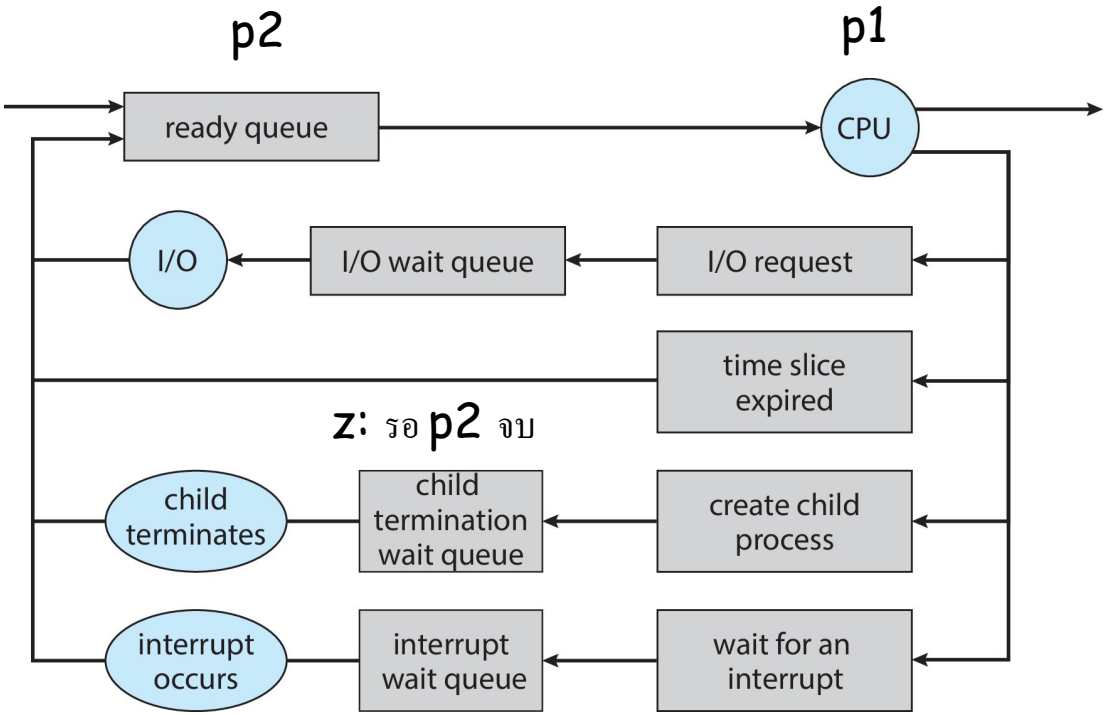
สมมุติว่า parent ของ parent wait() ลูก Zombie ของ parent ก็จะถูกกำจัด



ถ้าภายหลัง child process (orphan) นี้จบ init จะกำจัด zombie ทันที

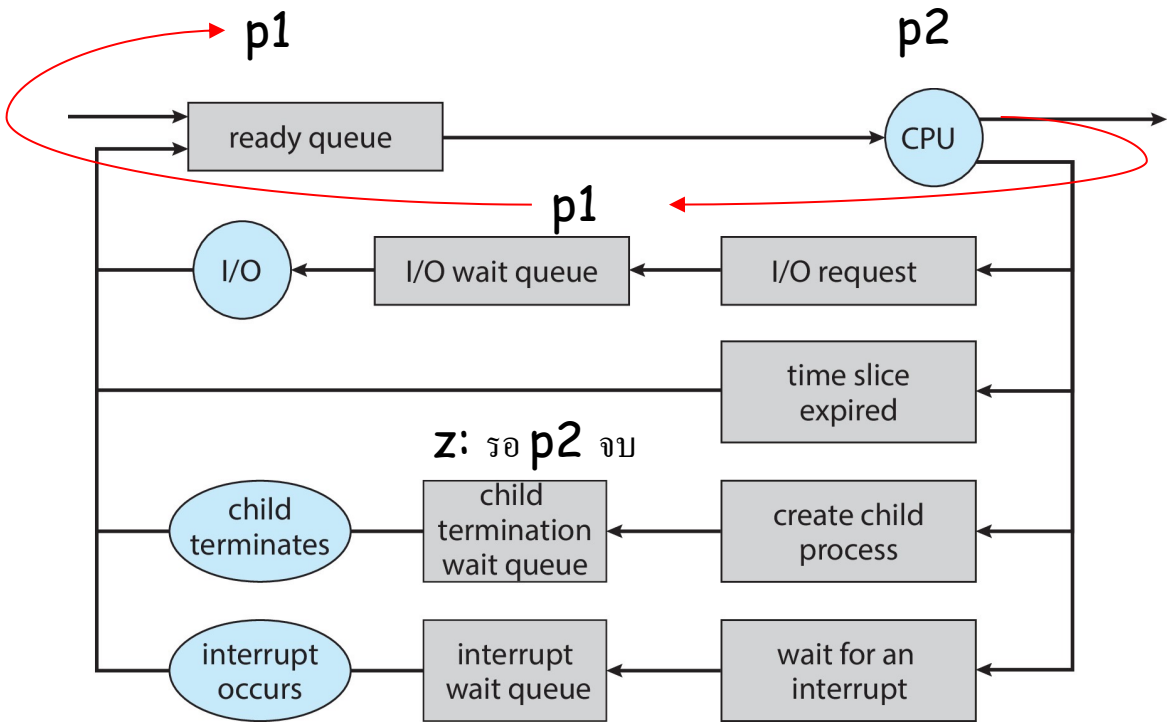
Revisit: Process Scheduling

1. เมื่อ P1 fork P2 แล้ว OS จะให้ P2 รอใน ready queue
   a. OS จะเอาชื่อ p1 ไปใส่ใน child termination wait queue
   b. เมื่อ p2 จบ OS จะเก็บข้อมูลการจบ และเช็ค queue นี้ว่าจะต้องแจ้งใคร (parent) ให้เข้ามาดู



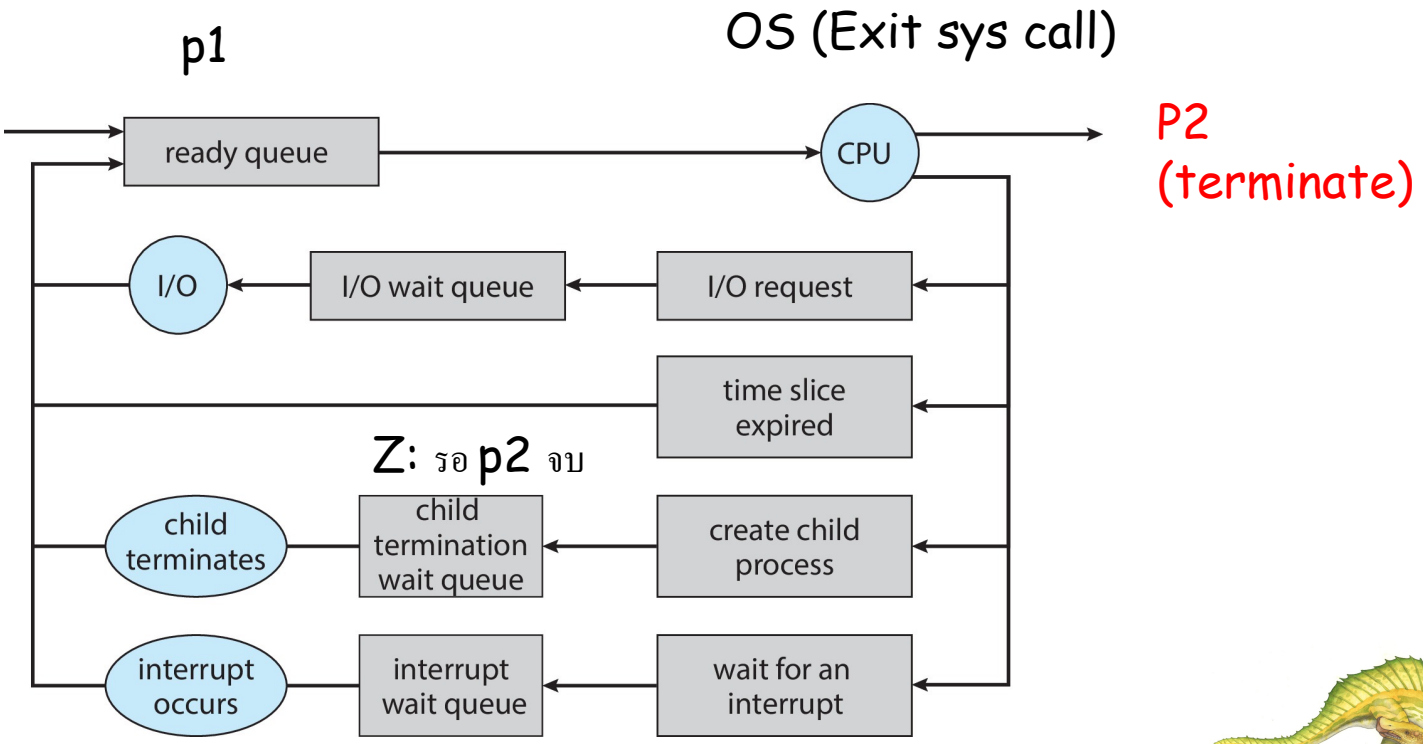
Revisit: Process Scheduling

2. สมมุติว่าในช่วงเวลาต่อมาp1 หมด time slice และp2 ได้เข้ามาประมวลผลซีพียู



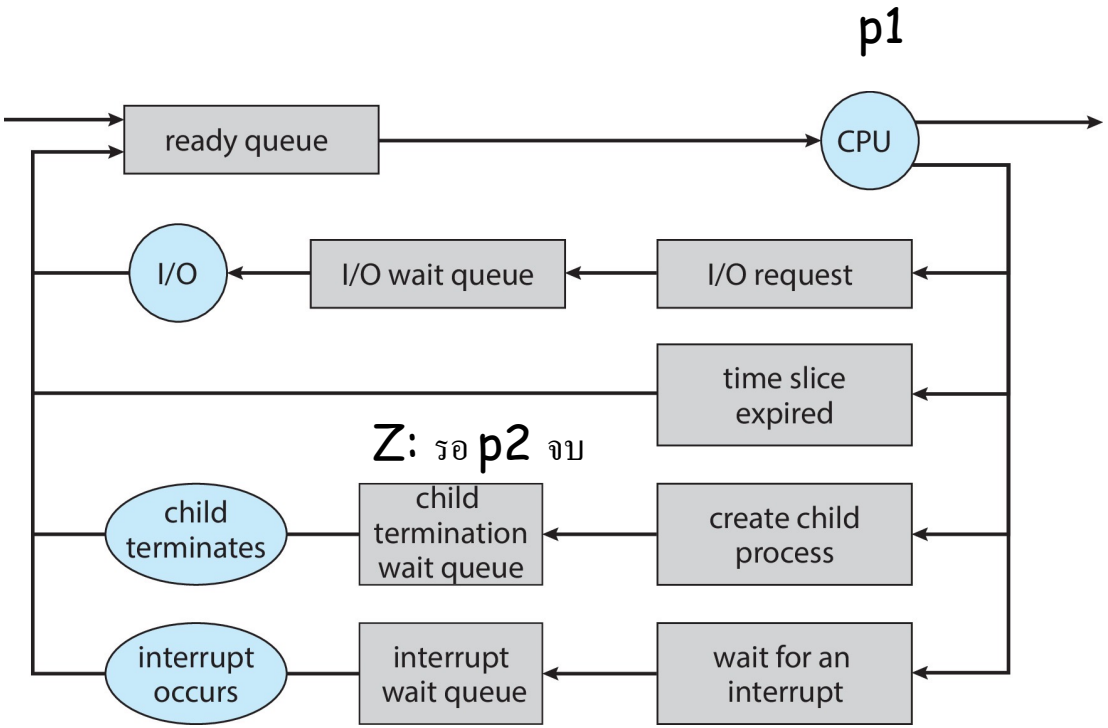
Revisit: Process Scheduling

3. p2 จบการทำงาน p2 จะเรียก exit() system call
4. OS เปลี่ยน State ของ p2 เป็น terminate
5. OS เก็บข้อมูลการจบของ p2 (และแจ้ง p1 ด้วย signal)



Revisit: Process Scheduling

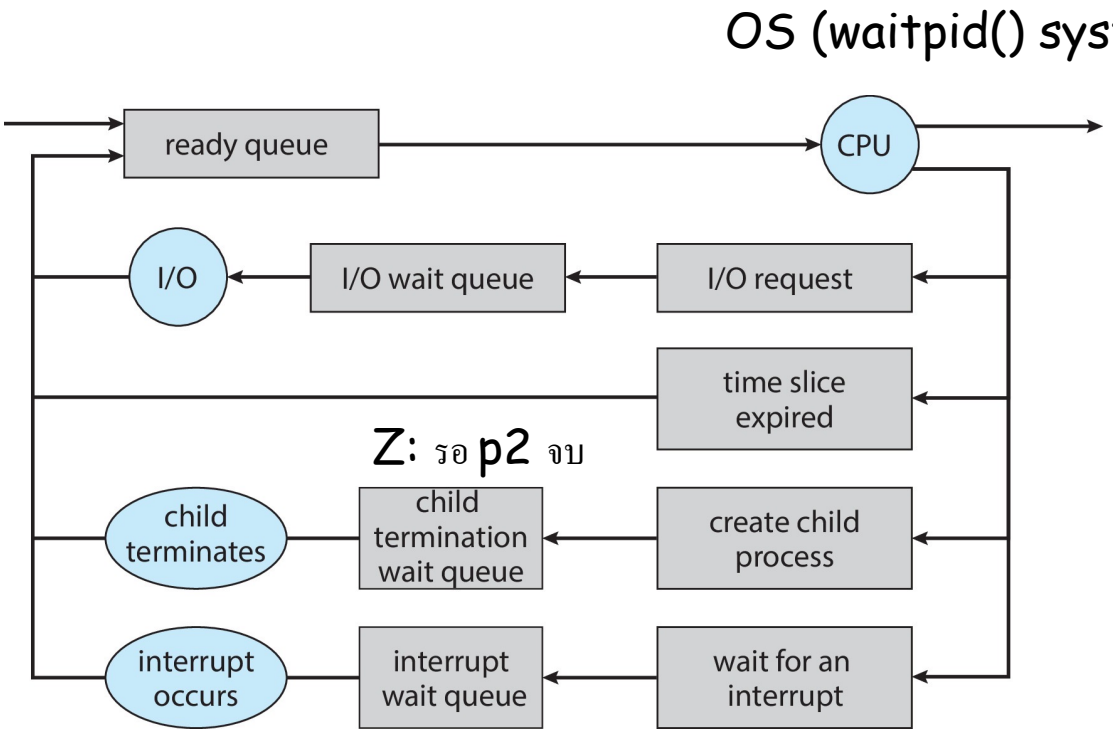
6. ต่อมาเมื่อ p1 ได้เข้ามาประมวลผลในซีพียู



P2 Zombie: OS เก็บข้อมูลการจบของ P2 ไว้ใน Memory ของ kernel

Revisit: Process Scheduling

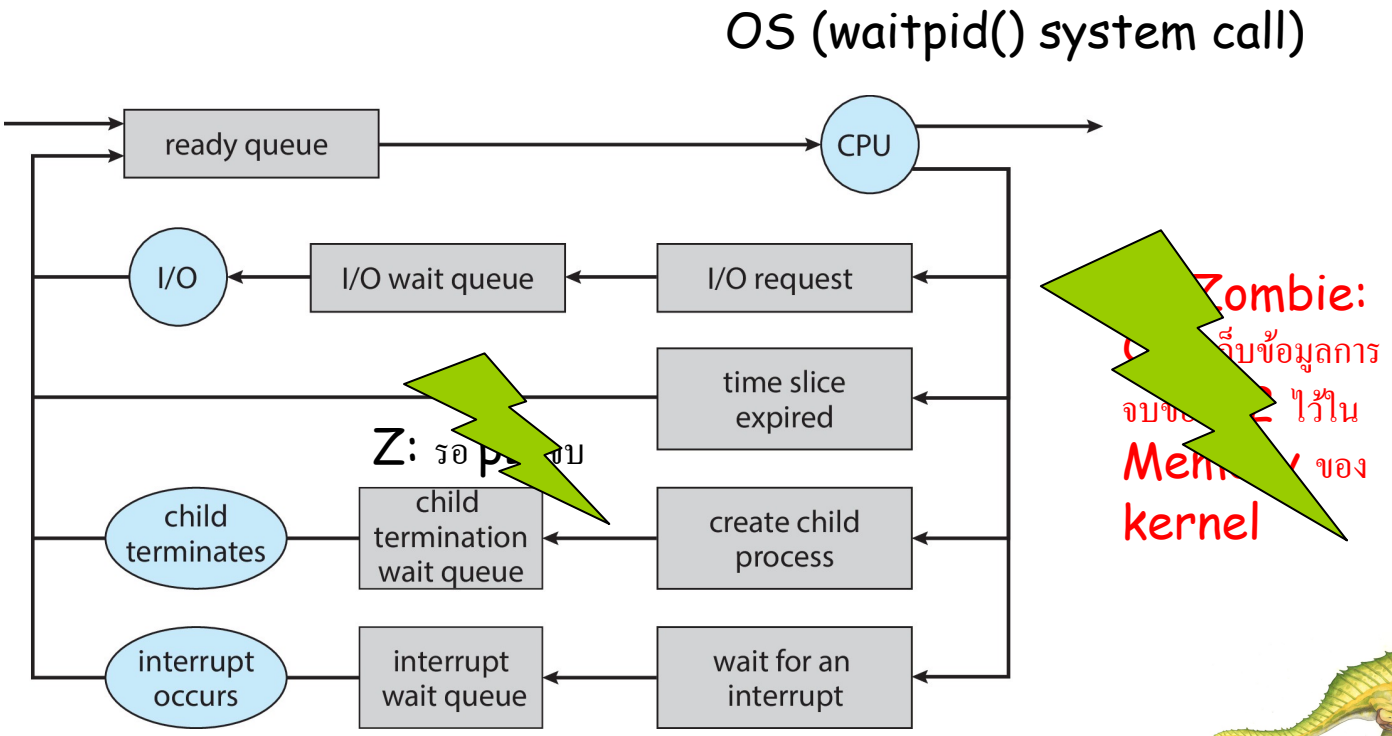
6. ต่อมาเมื่อ p1 ได้เข้ามาประมวลผลในซีพียู
7. p1 เรียก wait/waitpid system call (OS kernel)



P2 Zombie: OS เก็บข้อมูลการจบของ P2 ไว้ใน Memory ของ kernel

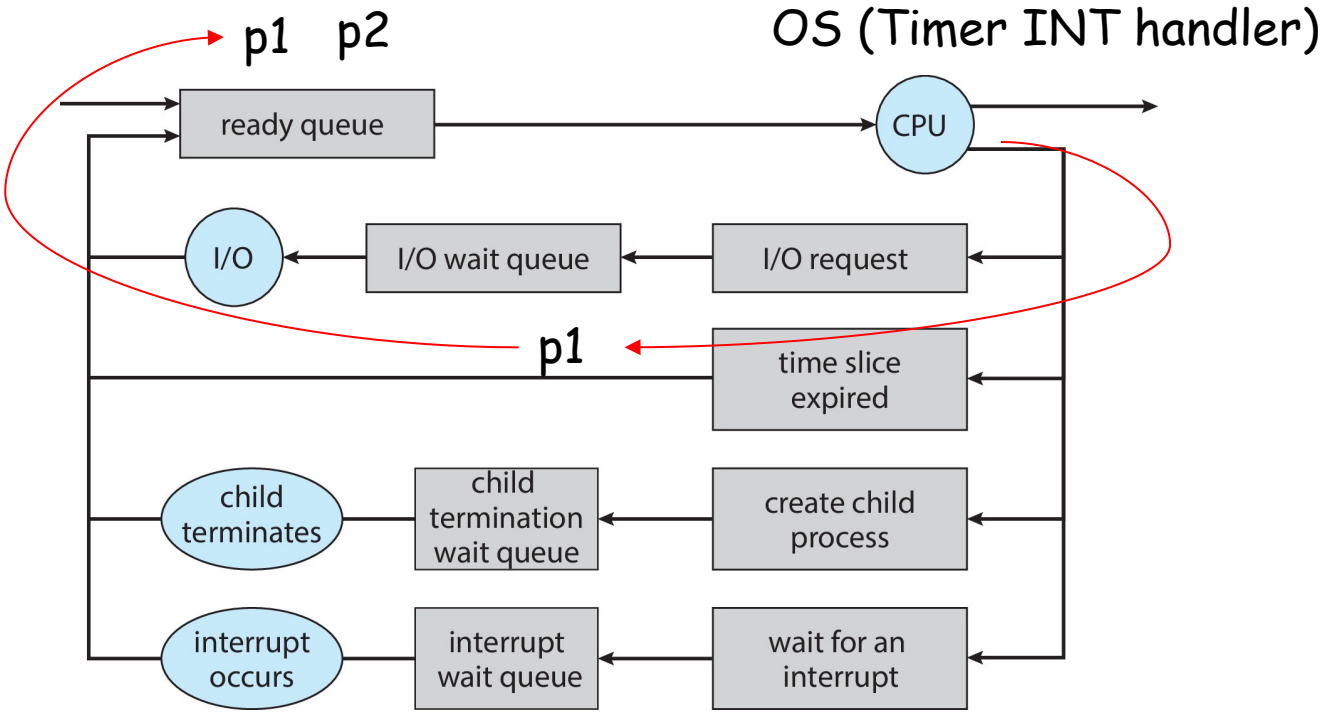
Revisit: Process Scheduling

8. OS เคลียร ข้อมูล p2's Zombie และ การรอของ p1 ใน child termination wait queue



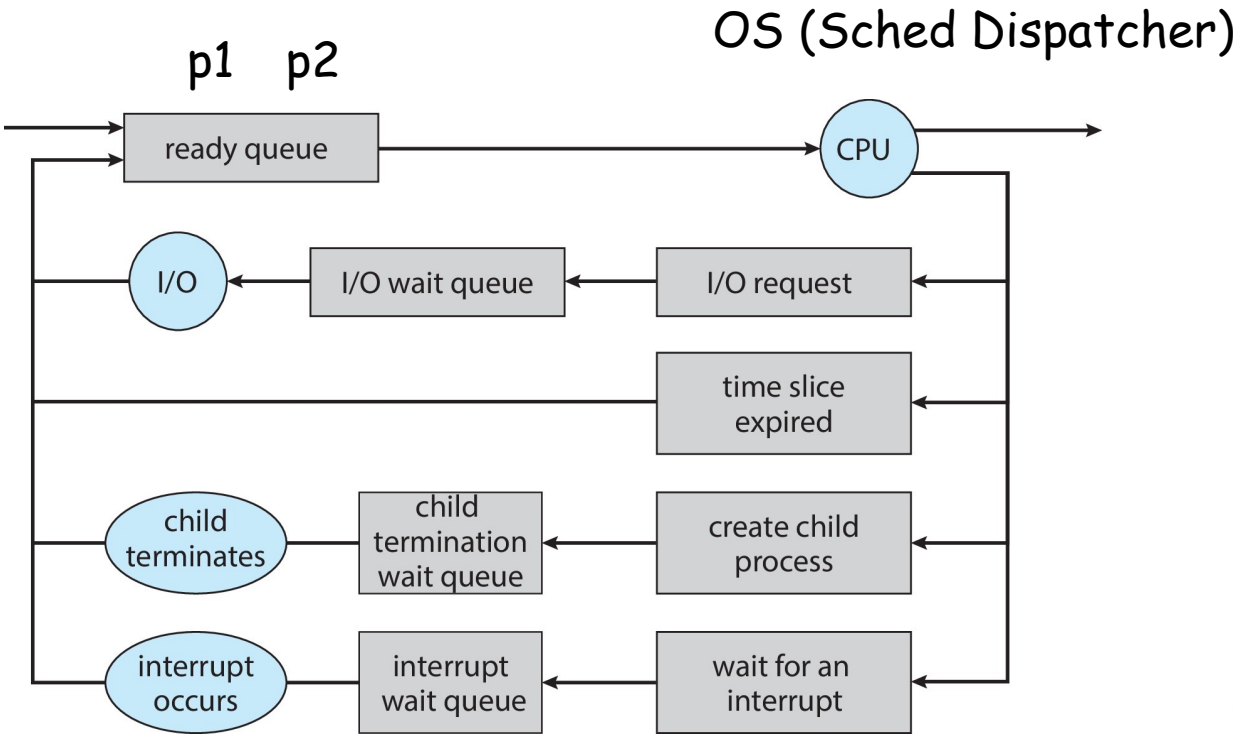
Revisit: Process Scheduling

1. สมมุติว่ามี p1 p2 ในระบบ และไม่ได้เป็น parent/child กัน
2. เมื่อเกิด Timer INT ซีพียูจะประมวลผล Timer INT handler
3. Handler จะนำ P1 ไปไว้ใน ready queue



Revisit: Process Scheduling

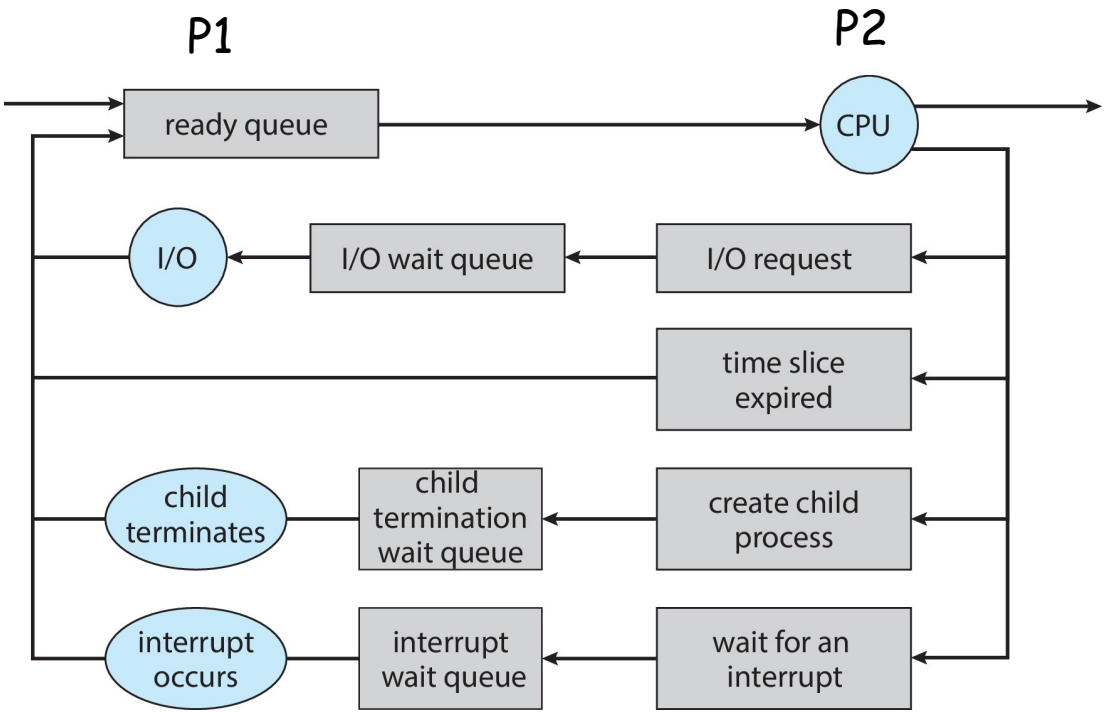
4. Handler เรียก Scheduler dispatcher ให้เลือก p2 เข้าใช้ซีพียู



Revisit: Process Scheduling

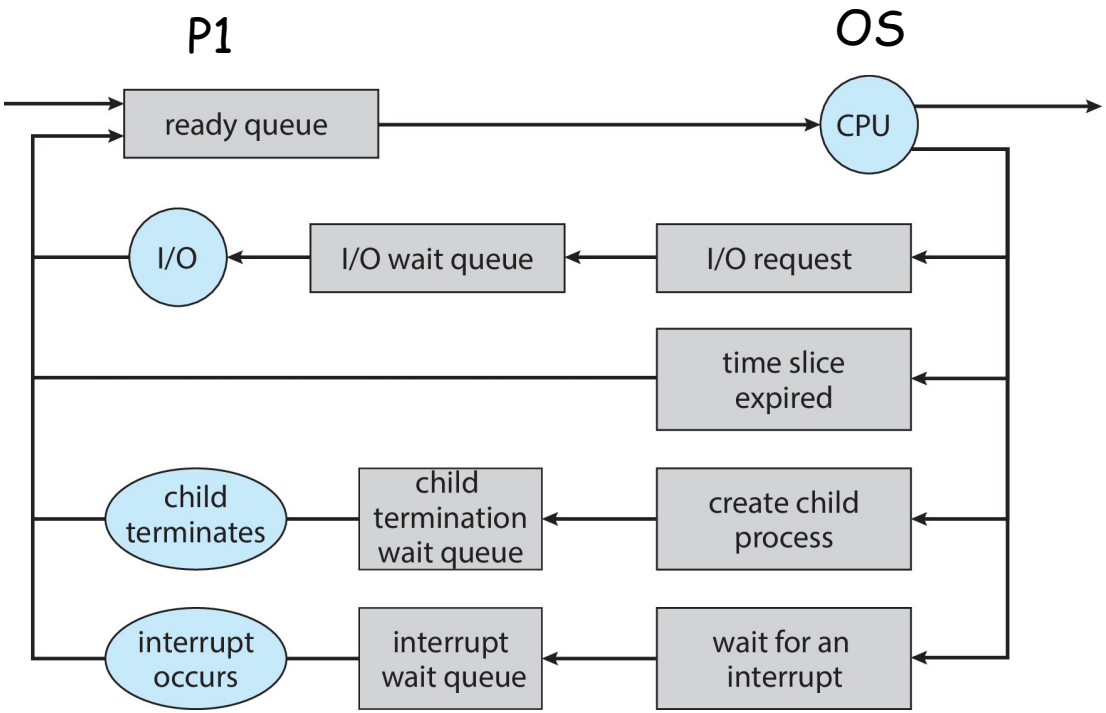
4. Handler เรียก Scheduler dispatcher ให้เลือก p2 เข้าใช้ซีพียู เปลี่ยน state ของ p2 เป็น running

5. Dispatcher ทำ Context switching ให้ p2 ประมวลผลบน CPU



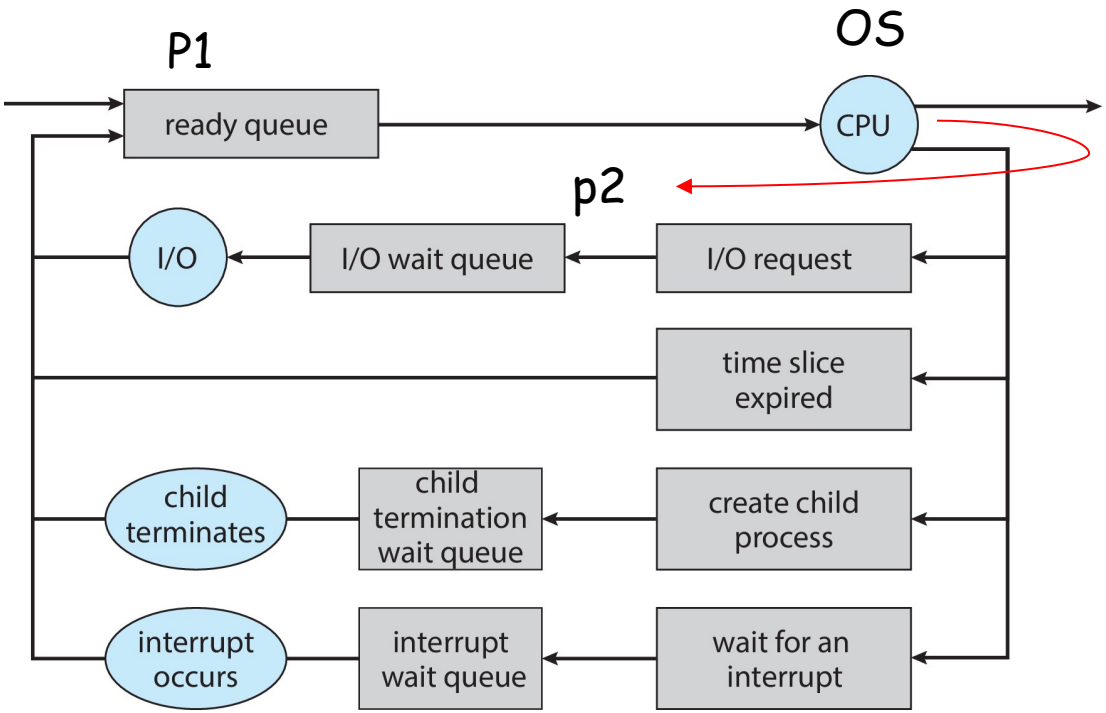
Revisit: Process Scheduling

6. p2 เรียก System call (OS kernel) เพื่อเขียนข้อมูลลง disk storage (พูดอีกอย่างคือ p2 สร้าง software interrupt)



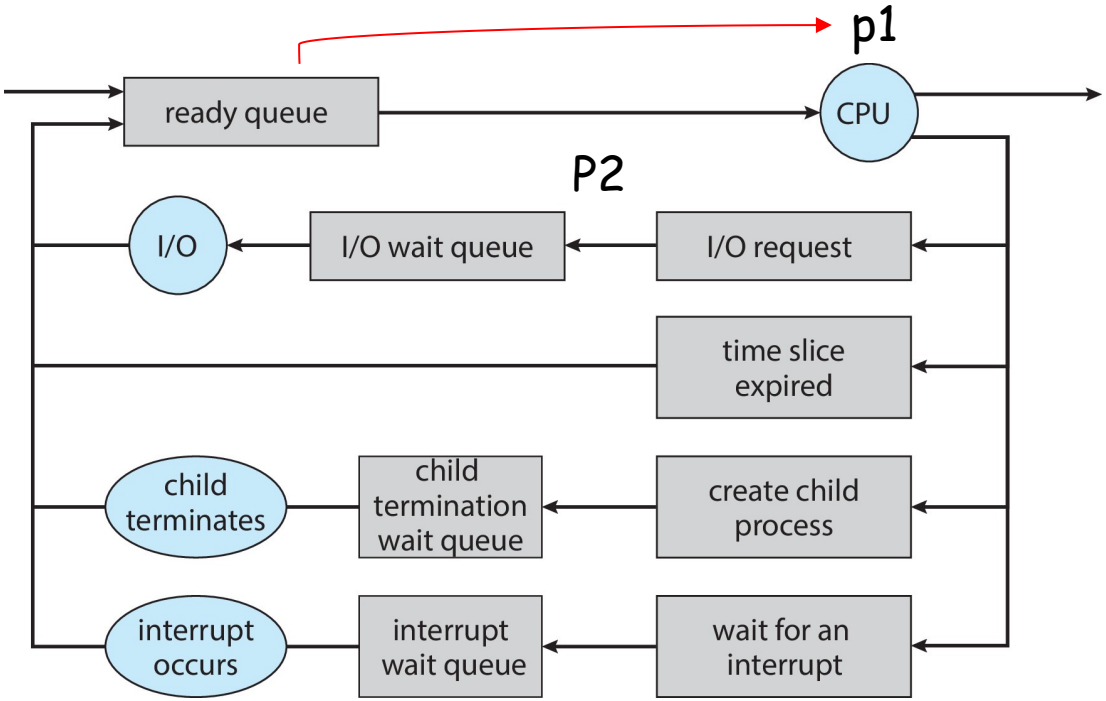
Revisit: Process Scheduling

6. p2 เรียก System call (OS kernel) เพื่อเขียนข้อมูลลง disk storage
7. OS เปลี่ยน state ของ p2 เป็น ”waiting” และนำ p2 ไปไว้ใน I/O request wait queue เพราะ I/O controller อาจยังไม่ว่างเพราะกำลังทำ I/O request ให้ process อื่นอยู่



Revisit: Process Scheduling

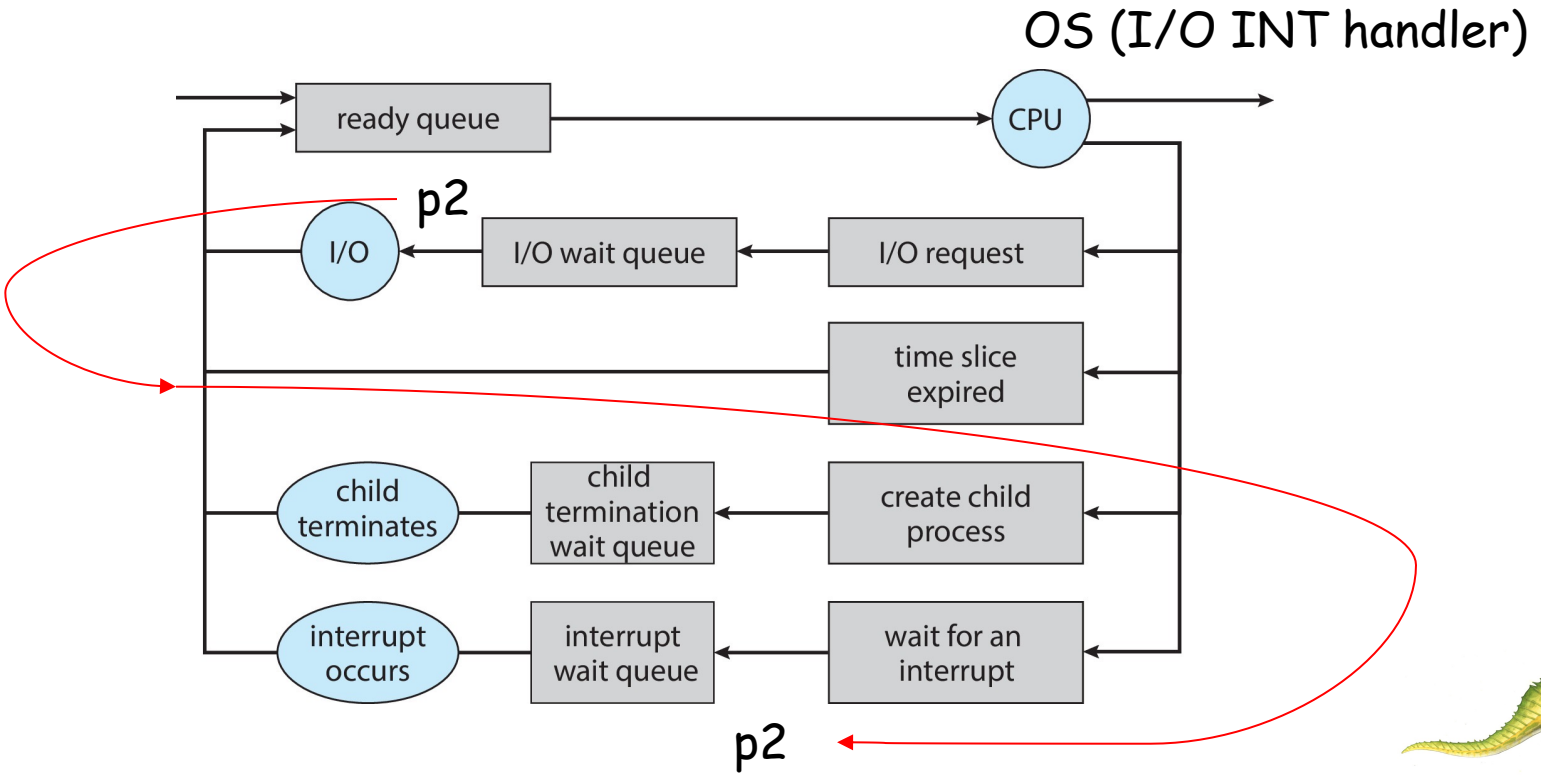
8. OS kernel เรียก Sched dispatcher เพื่อนำ process ใหม่ (p1) จาก ready queue มาใช้ซีพียู เปลี่ยน state ของ p1 เป็น running



Revisit: Process Scheduling

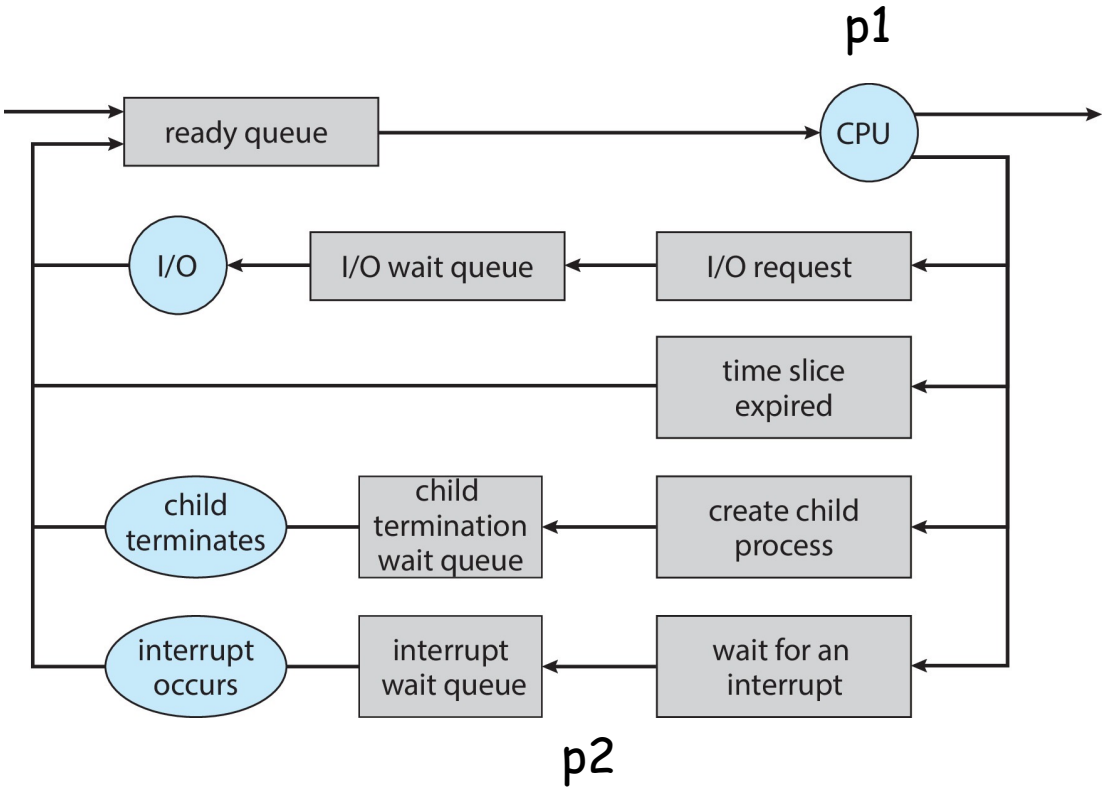
9. เมื่อ I/O controller พร้อมมันจะส่ง INT ไปให้ซีพียู
10. ซีพียูจะขัดจังหวะ p1 ชั่วคราวเพื่อรัน i/O INT handler
11. INT handler เรียก device driver เพื่อเขียนข้อมูลลง disk
12. INT handler ย้าย p2 ไปที่ interrupt waiting Queue

p1 (running *but interrupted*)



Revisit: Process Scheduling

13. เมื่อจบการทำงานของ I/O INT handler OS จะทำ CPU context switching เพื่อคืนซีพียูให้ประมวลผล p1 (state ของ p1 ยังคงเป็น running เหมือนเดิม)



สรุป

- หลักการเกี่ยวกับ Process
- โครงสร้างของ Process ใน Memory
- Process State Transition Diagram
- Scheduling: การกำหนดตารางขั้นตอนการเข้าใช้งาน CPU ของ Process
- Process Control Block และ Context Switching
- init process
- Orphan VS Zombie